**Лабораторная работа №6**

**“Тестирование производительности, профилирование приложений”**

**Цель работы:** Изучить основные методы тестирования производительности и профилирования многопоточных приложений.

**Продолжительность работы** - 4 часа.

**Оглавление**

[Нагрузочное тестирование или тестирование производительности 2](#_Toc497681771)

[Тестирование производительности (Performance testing) 2](#_Toc497681772)

[Стрессовое тестирование (Stress Testing) 2](#_Toc497681773)

[Объемное тестирование (Volume Testing) 2](#_Toc497681774)

[Тестирование стабильности или надежности (Stability / Reliability Testing) 3](#_Toc497681775)

[Определение целей тестирования производительности 3](#_Toc497681776)

[Основные тесты производительности 3](#_Toc497681777)

[Основные показатели (метрики) производительности 4](#_Toc497681778)

[1. Потребление ресурсов центрального процессора (CPU, %) 4](#_Toc497681779)

[2. Потребление оперативной памяти (Memory usage, Mb) 5](#_Toc497681780)

[3. Потребление сетевых ресурсов 5](#_Toc497681781)

[4. Работа с дисковой подсистемой (I/O Wait) 5](#_Toc497681782)

[5. Время выполнения запроса (request response time, ms) 6](#_Toc497681783)

[Профилирование 6](#_Toc497681784)

[Профилировщик Visual Studio 7](#_Toc497681785)

[1. Дискретный профилировщик 7](#_Toc497681786)

[2. Инструментированный профилировщик 10](#_Toc497681787)

[3. Профилировщик выделения памяти 11](#_Toc497681788)

[4. Профилировщик конкуренции 13](#_Toc497681789)

[Порядок выполнения лабораторной работы 15](#_Toc497681790)

# Нагрузочное тестирование или тестирование производительности

**Нагрузочное тестирование** или **тестирование производительности** – это автоматизированное тестирование, имитирующее работу определенного количества бизнес пользователей на каком-либо общем (разделяемом ими) ресурсе.

Основные виды нагрузочного тестирования:

* Тестирование производительности;
* Стрессовое тестирование;
* Объемное тестирование;
* Тестирование стабильности или надежности.

## ****Тестирование производительности**** (Performance testing)

Задачей тестирования производительности является определение масштабируемости приложения под нагрузкой, при этом происходит:

* измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций;
* определение количества пользователей, одновременно работающих с приложением;
* определение границ приемлемой производительности при увеличении нагрузки (при увеличении интенсивности выполнения этих операций);
* исследование производительности на высоких, предельных, стрессовых нагрузках.

## ****Стрессовое тестирование**** (Stress Testing)

Стрессовое тестирование позволяет проверить насколько приложение и система в целом работоспособны в условиях стресса и также оценить способность системы к регенерации, т.е. к возвращению к нормальному состоянию после прекращения воздействия стресса. Стрессом в данном контексте может быть повышение интенсивности выполнения операций до очень высоких значений или аварийное изменение конфигурации сервера. Также одной из задач при стрессовом тестировании может быть оценка деградации производительности, таким образом, цели стрессового тестирования могут пересекаться с целями тестирования производительности.

## ****Объемное тестирование**** (Volume Testing)

Задачей объемного тестирования является получение оценки производительности при увеличении объемов данных в базе данных приложения, при этом происходит:

* измерение времени выполнения выбранных операций при определенных интенсивностях выполнения этих операций;
* может производиться определение количества пользователей, одновременно работающих с приложением.

## ****Тестирование стабильности или надежности**** (Stability / Reliability Testing)

Задачей тестирования стабильности (надежности) является проверка работоспособности приложения при длительном (многочасовом) тестировании со средним уровнем нагрузки. Время выполнения операций может играть в данном виде тестирования второстепенную роль. При этом на первое место выходит отсутствие утечек памяти, перезапусков серверов под нагрузкой и другие аспекты, влияющие именно на стабильность работы.

## Определение целей тестирования производительности

Прямыми или косвенными целями любого тестирования, так или иначе затрагивающего вопросы производительности, является:

* определение "узких мест" системы (функций программно-аппаратного комплекса, обращение к которым приводит к наибольшему падению показателей производительности);
* определение лучшей архитектуры системы, выбор наилучшей платформы, средств и языков реализации;
* определение оптимального способа хранения файлов;
* оценка и оптимизация схемы базы данных в контексте повышения производительности;
* оценка максимальной и минимальной производительности системы и условий их достижения;
* определение характера увеличения времени отклика системы при увеличении нагрузки;
* определение максимального числа одновременно работающих пользователей, превышение которого делает использование системы невозможным;
* определение влияния конфигурации системы на производительность;
* оценка показателей масштабируемости системы;
* оценка соответствия сетевой инфраструктуры требованиям производительности.

# Основные тесты производительности

В рамках нагрузочного тестирования и тестирования производительности, как правило, выполняются следующие основные тесты.

**Тест на определение максимальных возможностей системы** (capacity test) позволяет определить т.н. "точку насыщения системы" (system saturation point) – уровень нагрузки, при котором дальнейшее наращивание числа пользователей ведёт к увеличению времени отклика системы либо ухудшению стабильности системы, но не к увеличению в единицу времени количества полезных операций, обработанных системой. Данный тест направлен на оценку производительности системы как аппаратно-программного комплекса, поскольку учитывает доступные аппаратные ресурсы и эффективность их использования.

Проведение нескольких тестов на определение максимальных возможностей системы с добавлением аппаратных ресурсов позволяет определить показатели масштабируемости (scalability) системы, которая определяется как способность приложения увеличивать производительность пропорционально добавлению аппаратных ресурсов системы.

**Низко-, средне- и высоконагруженная работа** (low-, mid-, high-load tests) – позволяет оценить время отклика (response time) системы в некоторых заданных диапазонах нагрузки. Данная информация может быть использована при составлении перечня требований к условиям эксплуатации системы.

**Тест на выживаемость** (longevity test) показывает способность системы работать длительное время под высокой нагрузкой. Одной из наиболее опасных проблем, выявляемых данным тестом, является утечка памяти и иное снижение эффективности использования аппаратных ресурсов из-за накапливающихся со временем ошибок в работе приложения.

**Тест "часа пик"** (rush hour test) позволяет оценить реакцию системы на резкое изменение нагрузки. Во время теста проверяется способность системы выдержать скачкообразное увеличение нагрузки во время "часа пик", а также способность системы вернуться к изначальным показателям производительности после завершения "часа пик" (восстановления исходных показателей нагрузки на систему). Такой тест позволяет выявить проблемы с синхронизацией выполнения отдельных участков кода, а также проблемы с управлением всеми видами межкомпонентного взаимодействия (в т.ч. сетевых и локальных соединений) на всех уровнях системы.

**Тест "точки рандеву"** (rendezvous point test) подразумевает такую настройку профиля нагрузки и поведения виртуальных пользователей, чтобы в некоторый момент все они одновременно выполняли одну и ту же операцию: как правило, синхронную операцию сохранения, записи, и т.п. В отличие от теста "часа пик" этот тест не подразумевает увеличения числа одновременно работающих с системой пользователей, а подразумевает исследование ситуации конкуренции пользователей за некоторые ресурсы, совместное использование которых не представляется возможным или сопряжено с повышенной нагрузкой на системные ресурсы. В частности, этот тест позволяет выявить проблемы с разделением ресурсов на уровне баз данных.

# Основные показатели (метрики) производительности

Одним из результатов, получаемых при нагрузочном тестировании и используемых в дальнейшем для анализа, являются показатели производительности приложения. Основные из них разобраны ниже.

## Потребление ресурсов центрального процессора (CPU, %)

Метрика, показывающая, сколько времени из заданного определённого интервала было потрачено процессором на вычисления для выбранного процесса. В современных системах важным фактором является способность процесса работать в нескольких потоках, для того, чтобы процессор мог производить вычисления параллельно. Анализ истории потребления ресурсов процессора может объяснять влияние на общую производительность системы потоков обрабатываемых данных, конфигурации приложения и операционной системы, мультипоточности вычислений, и других факторов.

## Потребление оперативной памяти (Memory usage, Mb)

Метрика, показывающая количество памяти, использованной приложением. Использованная память может делиться на три категории:

**Virtual** — объём виртуального адресного пространства, которое использует процессор. Этот объём не обязательно подразумевает использование соответствующего дискового пространства или оперативной памяти. Виртуальное пространство конечно и процесс может быть ограничен в возможности загружать необходимые библиотеки.

**Private** — объём адресного пространства, занятого процессом и не разделяемого с другими процессами.

**Working Set** — набор страниц памяти, недавно использованных процессом. В случае, когда свободной памяти достаточно, страницы остаются в наборе, даже если они не используются. В случае когда, свободной памяти остается мало, использованные страницы удаляются.

При работе приложения память заполняется ссылками на объекты, которые, в случае неиспользования, могут быть очищены специальным автоматическим процессом, называемым «сборщиком мусора» (англ. *Garbage Collector*). Время затрачиваемое процессором на очистку памяти таким способом может быть значительным, в случае, когда процесс занял всю доступную память (в Java — так называемый «постоянный Full GC») или когда процессу выделены большие объёмы памяти, нуждающиеся в очистке. На время, требующееся для очистки памяти, доступ процесса к страницам выделенной памяти может быть заблокирован, что может повлиять на конечное время обработки этим процессом данных.

## Потребление сетевых ресурсов

Эта метрика не связана непосредственно с производительностью приложения, однако её показатели могут указывать на пределы производительности системы в целом.

|  |
| --- |
| **Пример:**  *Серверное приложение обрабатывая запрос пользователя, возвращает ему видео-поток, используя сетевой канал в 2 мегабит. Требование гласит, что сервер должен обрабатывать 5 запросов пользователей одновременно.*  *Нагрузочное тестирование показало, что эффективно сервер может предоставлять данные только 4 пользователям одновременно, так как мультимедиа-поток имеет битрейт в 500 килобит. Очевидно, что предоставление этого потока 5 пользователям одновременно невозможно в силу превышения пропускной способности сетевого канала, а значит, система не удовлетворяет заданным требованиям производительности, хотя при этом потребление ей ресурсов процессора и памяти может быть невысоким.* |

## Работа с дисковой подсистемой (I/O Wait)

Работа с дисковой подсистемой может значительно влиять на производительность системы, поэтому сбор статистики по работе с диском может помогать выявлять узкие места в этой области. Большое количество чтений или записей может приводить к простаиванию процессора в ожидании обработки данных с диска и в итоге увеличению потребления CPU и увеличению времени отклика.

## Время выполнения запроса (request response time, ms)

Время выполнения запроса приложением остаётся одним из самых главных показателей производительности системы или приложения. Это время может быть измерено на серверной стороне, как показатель времени, которое требуется серверной части для обработки запроса; так и на клиентской, как показатель полного времени, которое требуется на сериализацию / десериализацию, пересылку и обработку запроса. Надо заметить, что не каждое приложение для тестирования производительности может измерить оба этих времени.

# Профилирование

***Профилирование*** — это сбор характеристик программы, таких как время выполнения отдельных ее фрагментов (функций, строк).

Существует два основных типа инструментов для определения полного профиля приложения: *профилирование* и *трассировка*. Важно различать профилирование и трассировку, так как они несхожие, но взаимодополняющие инструменты. Профилирование приложения обычно означает агрегирование или суммирование статистики применения во время работы. Трассировка же собирает данные, архив событий, в то время как выполняется приложение, и представляет их как динамику изменений. Профилирование обычно предоставляет довольно небольшое количество данных, в то время как трассировка может предоставлять большое количество информации.

Трассировка предоставляет данные о проведённом времени в определённом режиме или о наборе вложенных циклов. Профилирование же выходит за рамки этого и проводит мониторинг системы во время работы приложения, которая действительно контролирует "события", происходящие в системе. Например, вы можете измерить количество различных промахов кэша или хитов, промахи буфера ассоциативной трансляции (TLB), ошибочность прогнозирования ветвления, количества команд, количество памяти нагрузок / блоков памяти, количество операций с плавающей точкой в секунду и так далее. Как правило, эти данные представлены в виде динамики изменений (то есть участка в зависимости от времени).

Инструменты мониторинга могут быть поделены на две главные группы — встраиваемые блоки и отдельные независимые программы.

Для первой категории существуют такие инфраструктуры как ETW (Event Tracing for Windows) и счетчики производительности. Для второй – библиотеки, которые встраиваются в ваш код, например, New Relic, и полностью готовые системы мониторинга, которые не требуют вмешательства программистов.

# Профилировщик Visual Studio

1. **Дискретный профилировщик**

Он периодически прерывает работу приложения и сохраняет информацию о стеке вызовов для каждого процессора, где в настоящий момент протекает выполнение приложения или потока.

Сбор информации с использованием профилировщика Visual Studio обходится достаточно недорого, и если интервал следования событий достаточно широк (по умолчанию он составляет 10 000 000 тактов часов CPU), накладные расходы в общем времени выполнения приложения будут составлять менее 5%. Кроме того, дискретный подход к сбору информации позволяет подключиться к выполняющемуся процессу, собрать данные в течение некоторого времени и затем отключиться от процесса, чтобы проанализировать их. Из-за этой особенности, выявление проблем производительности рекомендуется начинать с поиска участков программы, оказывающих самую большую нагрузку на процессор - методов, на выполнение которых тратится больше всего процессорного времени.

По окончании сеанса сбора информации, профилировщик генерирует сводные таблицы, где каждый метод характеризуется двумя значениями: **числом исключительных попаданий** (*exclusive samples*), когда в момент отбора очередной порции данных на CPU выполнялся данный метод, и **числом включительных попаданий** (*inclusive samples*), когда выполнялся данный метод или вызванный им. Методы с большим количеством исключительных попаданий оказывают самую большую нагрузку на процессор; методы с большим количеством включительных попаданий не используют процессор непосредственно, но вызывают другие методы, суммарно занимающие значительную часть процессорного времени. (Например, в однопоточных приложениях метод Main() будет иметь 100% включительных попаданий.)

Запуск дискретного профелировщика:

1. В Visual Studio выберите пункт меню Analyze -> Launch Performance Wizard (Анализ --> Запустить мастер производительности).
2. На первой странице мастера отметьте радиокнопку CPU sampling (Выборка циклов ЦП (рекомендуется)) и щелкните на кнопке Next.

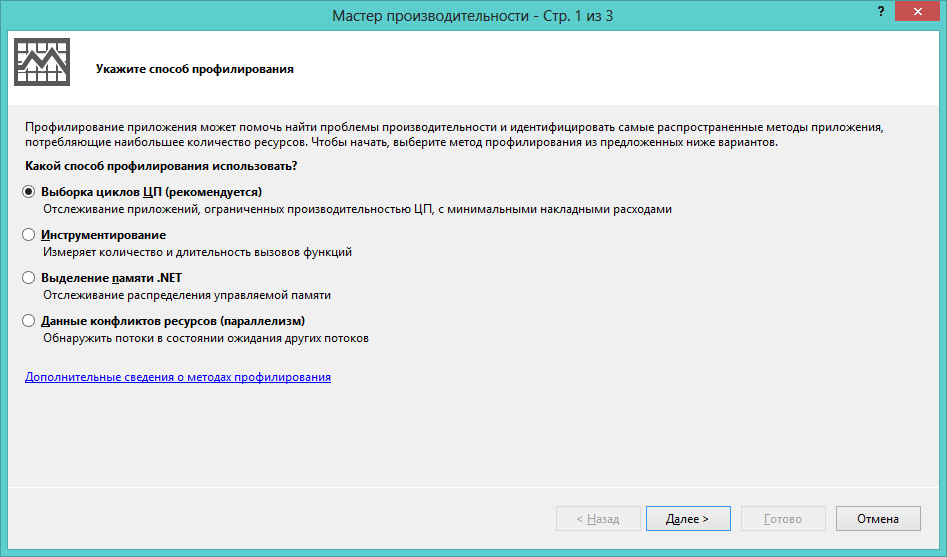


Рисунок 1. Выбор способа профилирования

1. Если проект для профилирования загружен в текущем решении, щелкните на радиокнопке One or more available projects (Один или несколько доступных проектов) и выберите проект из списка. В противном случае щелкните на радиокнопке An executable (.EXE file) (Исполняемый файл). Щелкните на кнопке Next.
2. Если на предыдущем шаге вы выбрали параметр An executable (.EXE file), укажите профилировщику путь к выполняемому файлу и аргументы командной строки, если они необходимы, затем щелкните на кнопке Next. (Если у вас нет собственного подходящего приложения, используйте JackCompiler.exe из исходников.)
3. Оставьте флажок Launch profiling after the wizard finishes (Запустить профилирование после завершения работы мастера) отмеченным и щелкните на кнопке Finish (Готово).
4. Если среда разработки Visual Studio была запущена не с привилегиями администратора, вам будет предложено повысить привилегии профилировщика.
5. Когда приложение завершит выполнение, откроется отчет профилирования. Используйте раскрывающийся список Current View (Текущее представление) для навигации между представлениями, отображающими информацию, собранную в процессе профилирования.

По окончании сеанса профилирования можно также воспользоваться обозревателем производительности - Performance Explorer, чтобы запустить его, выберите пункт меню Analyze -> Windows -> Performance Explorer (Анализ --> Окна --> Обозреватель производительности). С его помощью можно изменить параметры профилирования (например, выбрать другой интервал опроса или установить другой критерий), изменить целевой файл и сравнить результаты разных сеансов профилирования.

На рисунке ниже показано окно профилировщика с результатами, где можно видеть ветви в стеке вызовов, на выполнение которых было затрачено больше всего времени, и функции с наибольшим количеством исключительных попаданий:

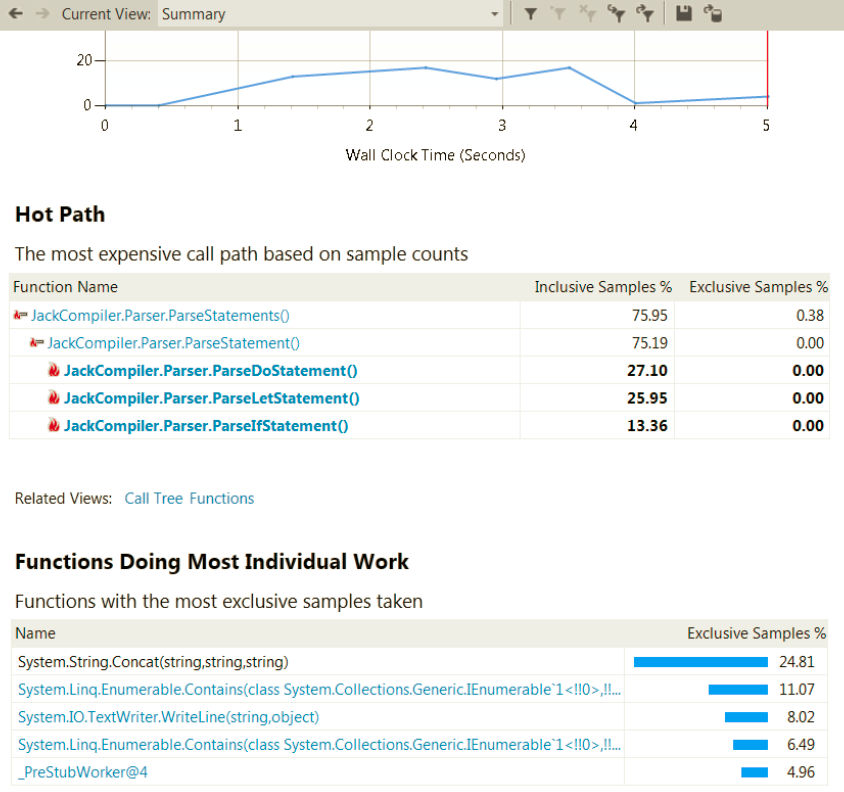


Рисунок 2. Результаты дискретного профилирования из представления Summary

На следующем рисунке показан подробный отчет, где перечислено несколько методов, на выполнение которых затрачено больше всего процессорного времени (имеющих наибольшее количество исключительных попаданий):

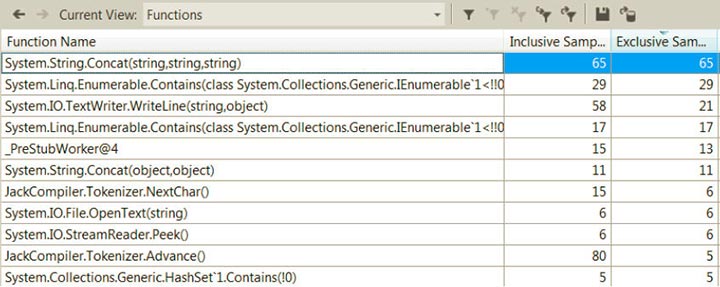


Рисунок 3. Результаты дискретного профилирования из представления Functions

Как видно, функция System.String.Concat имеет, как минимум, в два раза больше попаданий, чем любая другая функция. Если выполнить двойной щелчок на методе в списке, откроется окно детализации со строками исходного кода в приложении, в которых обнаружено наибольшее число попаданий:

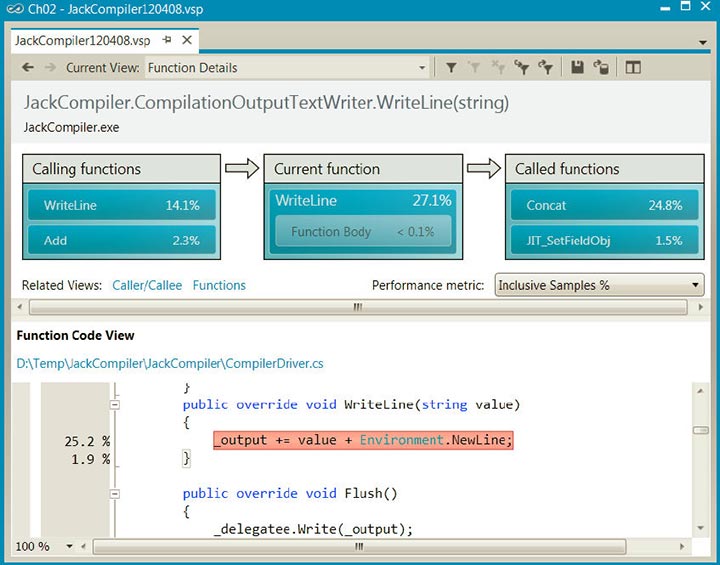


Рисунок 4. Результаты дискретного профилирования из представления Function Details

Результаты дискретного профилировщика следует рассматривать не как точную картину распределения процессорного времени, а как общую схему, где выделены возможно узкие места.

Помимо количества исключительных/включительных попаданий для каждого метода, профилировщик Visual Studio предлагает массу другой ценной информации. Например, представление Call Tree (Дерево вызовов) отображает иерархию вызовов методов в приложении, представление Lines (Строки) отображает информацию о попаданиях на уровне строк, а представление Modules (Модули) группирует методы по сборкам, и может помочь быстро выяснить, в каком направлении двигаться в поисках узких мест.

Так как при дискретном мониторинге требуется, чтобы поток выполнения в приложении был активен в моменты сбора информации, нет никакого способа получить информацию о потоках, простаивавших в ожидании выполнения операций ввода/вывода или на блокировках механизмов синхронизации. Дискретный мониторинг отлично подходит для анализа приложений, занимающихся преимущественно вычислениями, но для приложений, основная работа которых заключается в выполнении операций ввода/вывода, необходимы иные подходы, опирающиеся на использование других, более глубинных механизмов профилирования.

1. **Инструментированный профилировщик**

Он позволяет измерить общее время выполнения, а не только процессорное время, хорошо подходит для профилирования приложений, выполняющих большое количество операций ввода/вывода или интенсивно использующих механизмы синхронизации.

В режиме инструментированного профилирования профилировщик изменяет целевой выполняемый файл и внедряет в него код, выполняющий измерения и сообщающий профилировщику точную информацию о времени выполнения и количестве вызовов каждого оцениваемого таким способом метода.

Инструментированный профилировщик генерирует отчет с представлением Summary (Сводка), содержащим информацию о наиболее затратных ветвях в стеке вызовов и отдельных функциях, на выполнение которых тратится больше всего времени. Информация в отчете будет основана на точном хронометраже, проводимом инструментированным кодом.

На рисунке ниже показано представление Functions, где доступны включительное и исключительное время, измеренные в миллисекундах, а также счетчики вызовов функций:

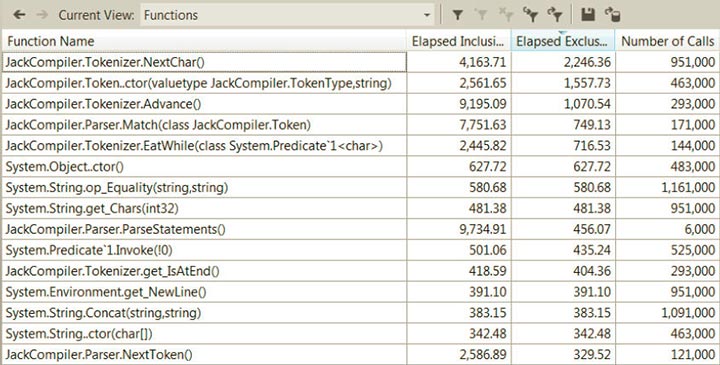


Рисунок 5. Результаты инструментированного профилирования из представления Functions

Инструментированное профилирование выглядит более точным, но на практике рекомендуется использовать дискретное профилирование, если приложение в основном решает вычислительные задачи. Прием инструментирования имеет ограниченную гибкость из-за необходимости изменять выполняемый файл приложения перед запуском и невозможности подключить профилировщик к уже запущенному процессу. Кроме того, инструментированное профилирование имеет немалые накладные расходы - объем выполняемого кода существенно увеличивается и в процессе выполнения часть времени затрачивается на отбор информации в точках входа и выхода из методов.

Как всегда, будет ошибкой безоговорочно доверять результатам инструментированного профилирования. Разумеется, количество вызовов того или иного метода не изменится из-за того, что приложение подвергается инструментированному профилированию, но информация о времени все еще может существенно искажаться из-за накладных расходов, несмотря на все попытки профилировщика сместить все дорогостоящие вычисления в конец. При внимательном подходе, дискретное и инструментированное профилирование могут помочь понять, где приложение проводит больше всего времени, особенно если сопоставлять множество отчетов и обращать внимание на результаты, полученные в результате оптимизации.

1. **Профилировщик выделения памяти**

Профилировщик Visual Studio способен собирать информацию об операциях выделения памяти и жизненном цикле объектов (которые освобождаются сборщиком мусора) в обоих режимах, дискретном и инструментированном. В дискретном режиме профилировщик собирает информацию о выделении памяти в приложении в целом. В инструментированном режиме информация собирается только из инструментированных модулей.

Для того, чтобы запустить профилировщик выделения памяти, в мастере настройки профилировщика производительности нужно выбрать ***радиокнопку .NET memory allocation*** (Выделение памяти .NET). В конце сеанса профилирования, в представлении Summary (Сводка), будут показаны функции, выделившие памяти больше всего:

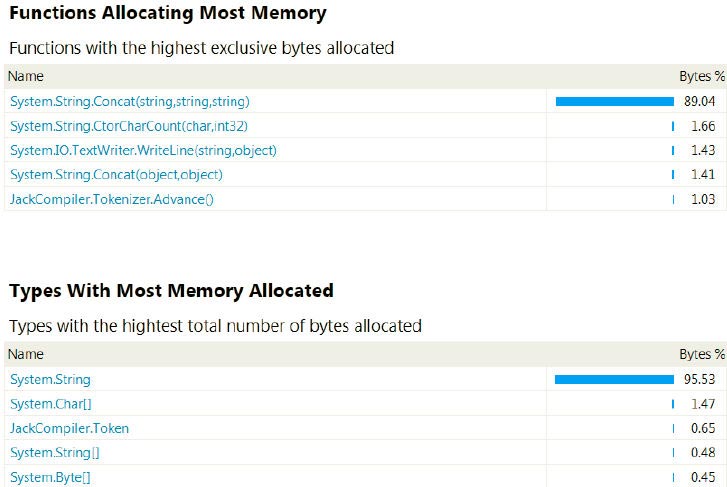


Рисунок 6. Результаты профилирования выделения памяти из представления Summar

В представлении Functions для каждого метода будет указано количество объектов и количество байтов памяти, выделенных методом (как обычно, включительные и исключительные значения). В представлении Function Details (Сведения о функции) будет представлена информация о вызывающих и вызываемых функциях, а также указаны строки кода с объемами выделенной ими памяти в поле слева:

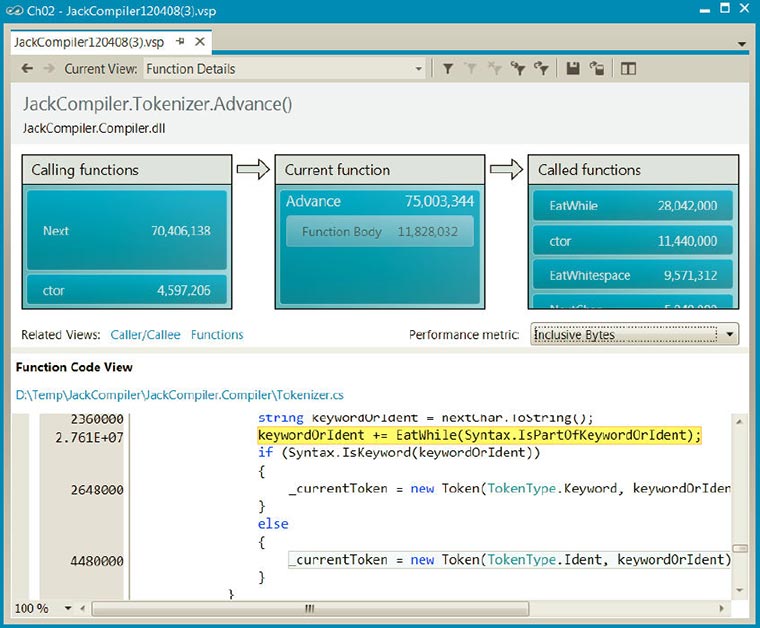


Рисунок 7. Результаты профилирования выделения памяти из представления Function Details

Но самая интересная информация содержится в представлении Allocation (Выделение), показывающем, какие ветви в стеке вызовов выделили памяти больше всего:

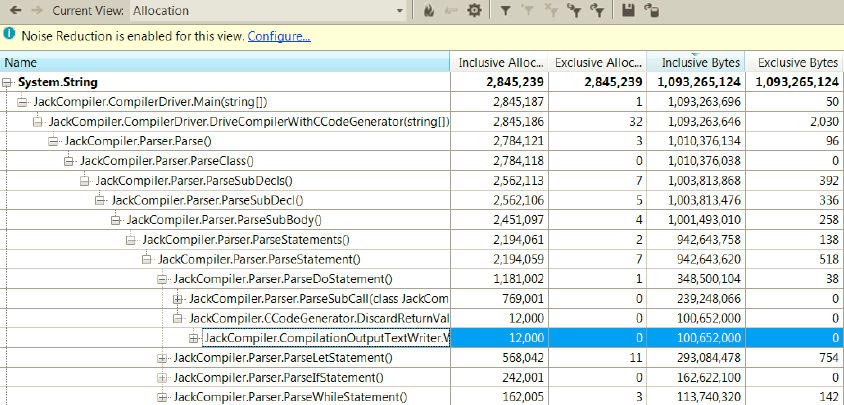


Рисунок 8. Результаты профилирования выделения памяти из представления Allocation

Представление Object Lifetime (Жизненный цикл объектов), сообщает, в каком поколении объекты были утилизированы. Это представление поможет увидеть, имеются ли объекты, пережившие слишком много циклов сборки мусора. На рисунке ниже можно видеть, что все объекты строк, созданные приложением (и занимающие более 1 Гбайта памяти!) были утилизированы в нулевом поколении, а это означает, что ни одному из них не удалось прожить дольше одного цикла сборки мусора:

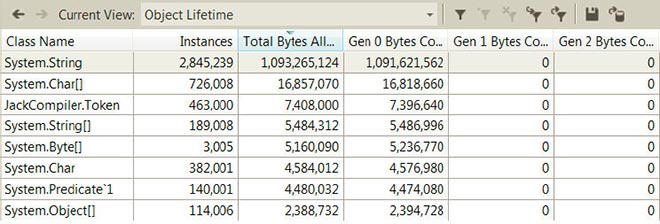


Рисунок 9. Результаты профилирования выделения памяти из представления Object Lifetime

1. **Профилировщик конкуренции**

Профилировщик Visual Studio, в режимах Concurrency (Данные конфликтов ресурсов (параллелизм)) и Concurrency Visualizer (Визуализатор параллелизма), использует события ETW для мониторинга производительности многопоточных приложений и предоставляет отчет с несколькими весьма полезными представлениями, упрощающими узкие места, отрицательно сказывающиеся на масштабируемости и производительности. Он имеет два режима работы.

В режиме **Concurrency** (Данные конфликтов ресурсов (параллелизм)) выявляются ресурсы, такие как управляемые блокировки, на которых потоки выполнения в приложении останавливаются в ожидании их освобождения. Первая часть отчета описывает сами ресурсы и потоки выполнения, которые были заблокированы на них - она поможет найти и устранить недостатки, мешающие масштабированию:

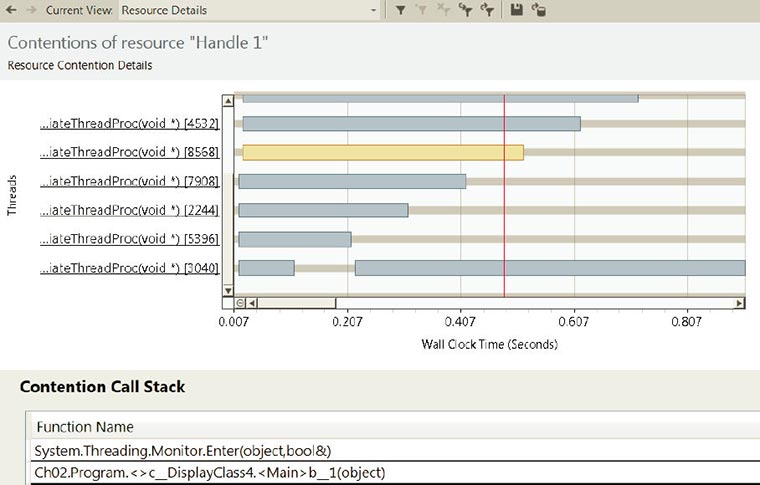


Рисунок 10. Результаты профилирования конкуренции из представления Resource Details

Как видно на рисунке, существует несколько потоков выполнения, ожидающих доступа к ресурсу. При выборе потока выполнения, в нижней части отображается его стек вызовов.

Другая часть отчета отображает информацию о конфликтах для конкретного потока выполнения, то есть перечень различных механизмов синхронизации, на которых поток выполнения вынужден был простаивать - она поможет уменьшить задержки на пути выполнения определенного потока.

Чтобы запустить профилировщик в этом режиме, нужно выбрать пункт меню Analyze --> Launch Performance Wizard (Анализ --> Запустить мастер производительности) и затем выбрать ***радиокнопку Concurrency*** (Данные конфликтов ресурсов (параллелизм)).

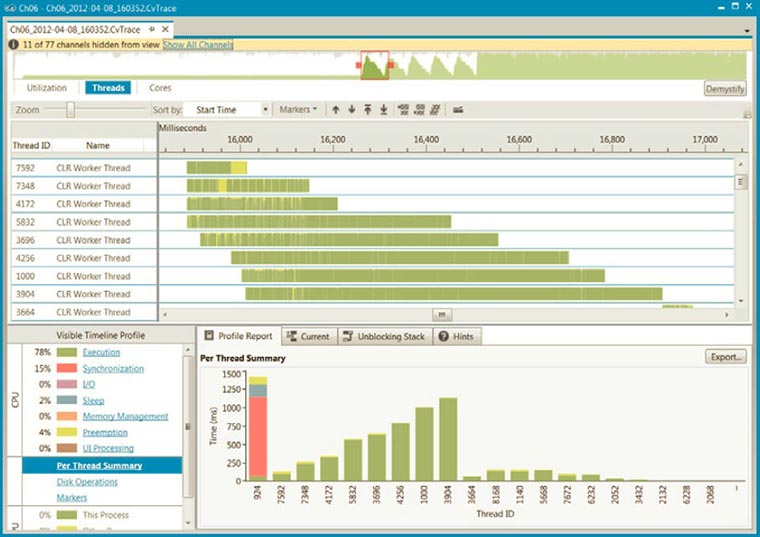


Рисунок 11. Сводная информация по потокам в режиме визуализация параллелизма

На основании этого отчета можно заключить, что работа распределяется между потоками неравномерно.

В режиме **визуализации параллелизма** отображается график выполнения всех потоков приложения, где цветом обозначается текущее состояние. Каждый поток имеет несколько переходных состояний - блокировка на операции ввода/вывода, ожидание на механизме синхронизации, выполнение - которые фиксируются профилировщиком, а также стек вызовов. Эти отчеты очень удобно использовать, чтобы понять особенности работы потоков выполнения и выявить причины низкой производительности, такие как чрезмерное количество потоков выполнения, недостаточное количество потоков выполнения и чрезмерное использование механизмов синхронизации.

В графике имеется также встроенная поддержка механизмов Task Parallel Library, таких как параллельные циклы (parallel loops) и механизмы синхронизации CLR. Чтобы запустить профилировщик в этом режиме выберите нужный пункт в подменю Analyze --> ***Concurrency Visualizer*** (Анализ --> Визуализатор параллелизма).

# Порядок выполнения лабораторной работы

1. Написать многопоточную программу в соответствии с вариантом (количество потоков должно соответствовать возможному количеству потоков для вычислительной машины).
2. Провести профилирование с помощью **дискретного профилировщика**. В отчет занести результаты из представлений Summary, Functions. Для функций, на выполнение которых затрачено больше всего процессорного времени, добавить в отчет информацию из представления Function Details. Для данных в каждом из этих представлений написать комментарии.
3. Провести профилирование с помощью **инструментированного профилировщика**. В отчет занести результаты из представлений Summary, Functions. Для функций, у которых наибольшее исключительное время, добавить в отчет информацию из представления Function Details. Для данных в каждом из этих представлений написать комментарии.
4. Провести профилирование с помощью **профилировщика выделения памяти**. В отчет занести результаты из представлений Summary, Functions. Для функций, которые выделили больше всего памяти (исключительное значение), добавить в отчет информацию из представления Function Details. Также в отчет включить информацию из представления Allocation, а именно какие ветви в стеке вызовов выделили памяти больше всего. Для данных в каждом из этих представлений написать комментарии.
5. Провести профилирование с помощью **профилировщика конкуренции**. Для режима Concurrency в отчет занести результаты из представления Resource Details, Threads Details. Для режима Concurrency Visualizer – сводную информацию. Добавить комментарии.
6. На основании полученной информации сделать вывод о причинах временных затрат, большого выделения памяти и т.п. при выполнении программы. Можно собрать информацию и о других событиях, если это требуется для обоснования вывода.

**Варианты заданий:**

1. Разложение каждого числа от 1 до N на простые множетели.
2. Перемножение N матриц размерности MxM.
3. Свой вариант задания, предварительно согласовав его с преподавателем.

**Вспомогательные материалы:**

1. <https://habrahabr.ru/post/97817/>
2. <http://professorweb.ru/my/csharp/optimization/level1/1_5.php>
3. <http://www.ishodniki.ru/art/art_progr/net/493.html>

**Ссылки:**

1. <https://professorweb.ru/my/csharp/optimization/level1/1_4.php>
2. <http://www.protesting.ru/testing/types/loadtesttypes.html>
3. <https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BF%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%BE%D0%B4%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D0%B8>
4. <https://habrahabr.ru/post/98361/>
5. <https://blogs.msdn.microsoft.com/developer-tools-rus/2013/04/23/143/>