Лабораторная работа №2: синхронизация и накладные расходы на поддержку многопоточности

Содержание

1. Цель лабораторной работы 1
2. Инструкция для выполнения лабораторной работы 1
  2.1. Изучение профилируемого приложения 1
  2.2. Профилирование приложения 2
  2.3. Изменение архитектуры приложения: организация пула потоков 4
  2.4. Выбор примитивов синхронизации 5
  2.5. Снижение частоты синхронизации 5
3. Контрольные вопросы 6
4. Задания для самостоятельной работы 6
5. Литература 6

1. Цель лабораторной работы

Целью настоящей лабораторной работы является изучение способов снижения накладных расходов на поддержку многопоточности. Под накладными расходами понимаются непроизводительные издержки на работу с потоками (время их создания, управления и уничтожения), а также время, затрачиваемое на работу с примитивами синхронизации (например, время с момента посылки сигнала до его получения). Кроме того, рассматриваются вопросы, связанные с выбором и использованием примитивов синхронизации.

2. Инструкция для выполнения лабораторной работы

2.1. Изучение профилируемого приложения

В настоящей лабораторной работе используется учебное приложение **ClientServer**, которое имитирует работу системы, имеющей клиент-серверную архитектуру. Особенность состоит в том, что для простоты клиент и сервер не представляют собой различные процессы, а реализованы в виде потоков одного процесса. Сценарий приложения следующий: клиент посылает запросы нескольких типов, а сервер принимает и обрабатывает их. Обработка производится в специально создаваемых для этого потоках.

Откройте проект **ClientServer**, последовательно выполняя следующие шаги:

- запустите приложение **Microsoft Visual Studio 2005**,
- в меню **File** выполните команду **Open→Project/Solution…**,
- в диалоговом окне **Open Project** выберите папку **C:\ITPLab\ClientServer**,
- дважды щелкните на файле **ClientServer.sln** или, выбрав файл, выполните команду **Open**.

После открытия проекта в окне **Solution Explorer** выберите проект **ClientServer** и дважды щелкните на файле исходного кода **ClientServer.cpp**, как это показано на рис. 1. После этих действий программный код, с которым предстоит работать, будет открыт в рабочей области **Microsoft Visual Studio**.

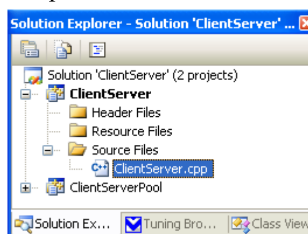

Рис. 1. Открытие файла **ClientServer.cpp**

В начале файла **ClientServer.cpp** объявлены две константы:

```
const int numRequests = 100;
const int numTypes = 4;
```

Первая из них указывает количество запросов от клиента к серверу, вторая показывает, сколько существует различных типов запросов. В данном случае клиент передаст серверу 100 запросов четырех

типов. Структура запроса в нашем приложении предельно проста – это целое число от 0 до **numTypes**. То есть клиент каждый раз посылает серверу число, которое и означает тип пришедшего запроса.

Кроме того, в приложении как глобальные переменные объявлены очередь и вектор:

```
queue<int>  requests;           //queue for requests
vector<int> requestsStatistics; //requests statistics
```

Очередь используется для передачи запросов от клиента к серверу, а вектор для сохранения статистики о пришедших запросах. Так, например, значение **requestsStatistics[0]** показывает количество пришедших запросов типа 0.

Ознакомьтесь с кодом приложения. Как можно видеть, для обработки каждого запроса сервер создает новый поток. Таким образом, приложение состоит из трех компонент:

- функция **main** – рабочая функция потока-клиента;
- функция **ServerThreadFunc** – рабочая функция потока-сервера;
- функция **HandlerPoolThreadFunc** – рабочая функция потока-обработчика.

Скомпилируйте и запустите приложение стандартными средствами **Microsoft Visual Studio**:

- кликните правой кнопкой мыши на проекте **ClientServer** и выберите в контекстном меню пункт **Build Only ClientServer** (рис. 2);
- в этом же меню выполните команду **Debug→Start new instance**.

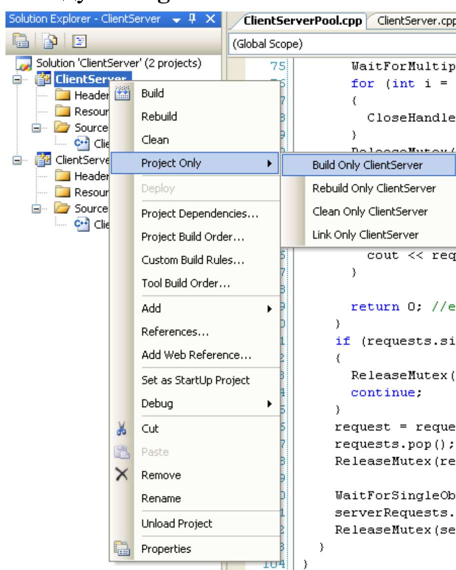

Рис. 2. Компиляция приложения

Убедитесь в правильности работы приложения по выводу на консоль.

2.2. Профилирование приложения

Запустите процесс профилирования в ITP, в его окне должны появиться результаты, как показано на рис.3.

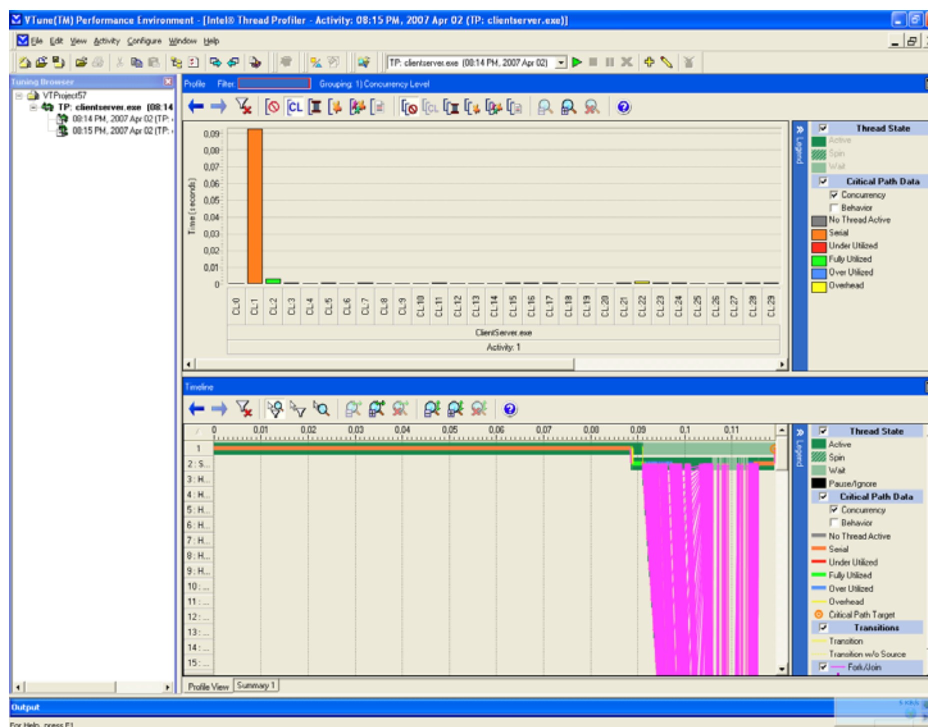

Рис. 3. Результаты профилирования приложения

Обратите внимание на область розового цвета в окне **Timeline**. При увеличении видим, что она образована стрелками, соответствующими вызовам **`fork/join`**. Это тревожный симптом, свидетельствующий о том, что в нашем приложении создается слишком много потоков. Ситуация эта плоха тем, что полезная работа, которую совершают потоки, не компенсирует затраты на их создание/уничтожение.

Действительно, для каждого запроса сервер создает новый поток, поэтому в нашем приложении существует **`numRequests`**+2 потока (сервер и клиент). Наш сервер, создавая и уничтожая потоки, тратит больше времени и потребляет больше системных ресурсов, чем если бы обрабатывал запросы самостоятельно. Кроме того, активные потоки также потребляют системные ресурсы, что может привести к нехватке оперативной памяти и значительному падению производительности.

Вообще говоря, работа многих серверов (web-серверы, серверы базы данных) связана с обработкой большого количества коротких запросов от какого-либо удаленного источника (клиента). При этом существует несколько распространенных вариантов архитектуры многопоточного сервера:

- обработка всех запросов в одном потоке;
- обработка каждого нового запроса в отдельном потоке;
- организация пула потоков.

Рассмотрим более подробно эти варианты.

2.2.1. Обработка всех запросов в одном потоке

Это решение подходит для тех случаев, когда количество запросов к серверу достаточно мало, и обращения к серверу происходят редко. При этом архитектура приложения очень проста, единственной трудностью является построение очереди входящих запросов, необходимой для предотвращения потери запросов при последовательной обработке.

Однако, этот подход крайне неэффективен при высокой частоте обращений к серверу. Если сервер производит обработку самостоятельно, время отклика будет очень большим (пропорционально сложности обработки). В итоге сервер будет не успевать обрабатывать все запросы. Кроме того, если сервер работает на двухъядерном узле, то мы будем наблюдать полную загрузку одного ядра и простой второго.

Именно поэтому в реальных приложениях этот подход используется крайне редко. Мы далее не будем касаться его.

2.2.2. Обработка каждого нового запроса в отдельном потоке

При такой схеме для каждого клиентского запроса создается отдельный поток. В рассматриваемом нами приложении реализован именно этот подход. Сервер имеет следующую архитектуру: основной поток

приложения ожидает поступления запросов от клиентов, и при поступлении нового создает поток, передавая клиентский запрос ему на обработку. Созданный поток выполняет соответствующую обработку и завершает свое существование.

Однако и этот подход имеет существенные недостатки:

- Частое создание и завершение потоков. Создание и завершение потока – весьма трудоемкая операция, требующая времени и ресурсов, поэтому уровень непроизводительных издержек очень высок.
- Нерегулируемое количество потоков. Это может привести к избыточному потреблению оперативной памяти, а также к резкому уменьшению свободной части виртуального адресного пространства процесса.
- Большое количество переключений контекстов рабочих потоков.

2.2.3. Организация пула потоков

Пул потоков предлагает решение перечисленных выше проблем. Стандартная схема организации приложения с пулом потоков выглядит следующим образом. Имеется основной поток приложения, ожидающий поступления клиентских запросов, и потоки, составляющие пул, которые создаются заранее или при поступлении первого запроса. При поступлении запроса главный поток выбирает свободный поток из пула и передает запрос ему на обработку, если же свободных потоков нет, то запрос помещается в очередь и ждет освобождения одного из потоков в пуле.

Положительный момент состоит в том, что потоки, однажды созданные, используются многократно, в результате чего издержки на создание и уничтожение потока малы по сравнению с его полезной работой. В итоге сокращается время обработки одного запроса, поскольку поток уже существует, когда прибывает очередной запрос.

Обычно имеет смысл ограничивать общее число рабочих потоков либо числом доступных процессоров (ядер), либо кратным ему (чтобы обеспечить равномерную загрузку ядер). Если потоки занимаются только вычислениями, их число обычно выбирается равным числу ядер. Если же потоки некоторое время находятся в состоянии ожидания (выполнение операций ввода-вывода), то число потоков может превышать число ядер. Обычно ограничивают число потоков удвоенным числом процессоров (ядер).

2.3. Изменение архитектуры приложения: организация пула потоков

Посмотрим, какие изменения произойдут в работе нашего приложения при введении пула потоков. Для этого в **Microsoft Visual Studio** в окне **Solution Explorer** выберите проект **ClientServerPool** и дважды щелкните на файле **ClientServerPool.cpp**.

В начале файла **ClientServerPool.cpp** объявлена новая константа, указывающая количество потоков в пуле:

```
const int numPoolThreads = 4;
```

Кроме того, вводится еще одна очередь, используемая сервером для хранения запросов при отсутствии свободных потоков в пуле:

```
queue<int> serverRequests; //request queue on server
```

Ознакомьтесь с кодом приложения. После этого запустите процесс профилирования в ITP, в результате чего рабочая область ITP должна принять вид, как показано на рис. 4.

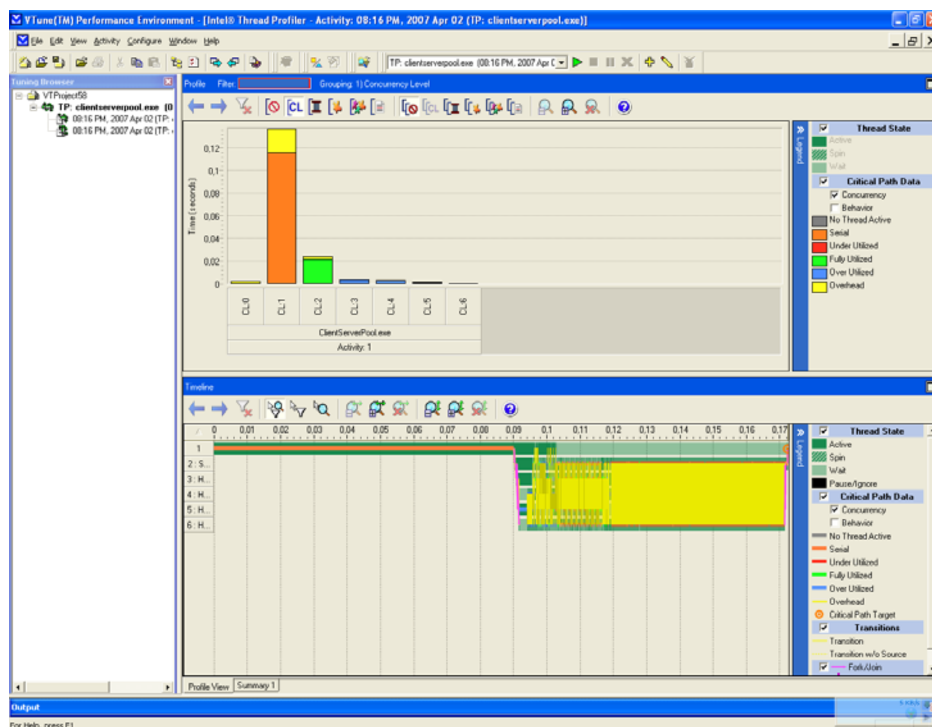

Рис. 4. Результаты профилирования приложения

При анализе можно заметить, что непроизводительные издержки значительно снизились, однако все еще присутствуют издержки, связанные с синхронизацией. Рассмотрим их подробнее в следующем разделе.

2.4. Выбор примитивов синхронизации

Основные издержки при синхронизации связаны с доступом потоков к глобальному массиву **`requestsStatistics`**, в котором хранится статистика запросов. При этом доступ происходит каждый раз в рамках критической области:

```
//enter to critical section (lock requestsStatisticsMutex mutex)
WaitForSingleObject(requestsStatisticsMutex,INFINITE);
//increment counter
requestsStatistics[serverRequest]++;
//release critical section (unlock requestsStatisticsMutex mutex)
ReleaseMutex(requestsStatisticsMutex);
```

Это сильно увеличивает время работы потоков-обработчиков, поскольку они вынуждены ожидать друг друга. Можно предложить следующие пути решения этой проблемы:

- использование объектов синхронизации пользовательского уровня (критическая секция), не использующих системные вызовы, вместо объектов уровня ядра операционной системы (мьютекс);
- предпочтение использования атомарных функций синхронизации ОС (**`AtomicIncrement`**, **`AtomicDecrement`**, etc);
- использование локальных переменных вместо глобальных.

Рассмотрим второй из этих подходов. Закомментируйте критическую область для доступа к **`requestsStatistics`** и раскомментируйте следующую строку в рабочей функции потока обработчика:

```
InterlockedIncrement(reinterpret_cast<long*>(&requestsStatistics[serverRequest]));
```

Проанализируйте произошедшие изменения и время работы приложения с помощью ITP.

2.5. Снижение частоты синхронизации

Можно заметить, что в нашем приложении синхронизация происходит слишком часто (большое количество стрелок желтого цвета). Это вызвано тем, что потоки работают с одними и теми же глобальными объектами, и вынуждены ждать друг друга. Эта проблема довольно часто встречается в многопоточных приложениях, и естественный подход к ее решению – снижение частоты синхронизации.

В нашем случае в качестве решения можно предложить следующий подход:

- на сервере вместо одной очереди сообщений создается **`numTypes`** очередей, в каждой из которых хранится запросы одинакового типа;

- сервер для каждого пришедшего запроса, определяет его тип и помещает его в соответствующую очередь сообщений;
- каждый поток обрабатывает сообщения только одного типа;
- каждый поток работает только со своим счетчиком, поэтому пропадает необходимость в синхронизации доступа к массиву `requestsStatistics`.

3. Контрольные вопросы

- В чем преимущество создания пула потоков перед обработкой всех запросов в одном потоке и обработкой каждого запроса в новом потоке?
- В чем недостатки использования примитивов синхронизации уровня ядра ОС?
- Как еще можно увеличить эффективность приложения **ClientServerPool**?

4. Задания для самостоятельной работы

- Реализуйте модель синхронизации с использованием критических секций, сравните ее по производительности с вариантами, использующими мьютексы и атомарные функции.
- Реализуйте схему синхронизации, изложенную в пункте 2.5.

5. Литература

1. «Developing Multithreaded Applications: A Platform Consistent Approach», Intel Corporation, March 2003.
2. «Threading Methodology: Principles and Practices», Intel Corporation, March 2003.
3. «Multi-Core Programming for Academia», Student Workbook, by Intel.
4. «Multi-Core Programming», book by Sh. Akhter and J. Roberts, Intel Press 2006.
5. «Intel® Thread Profiler. Getting Started Guide».
6. «Intel® Thread Profiler. Guide to Sample Code».
7. «Intel® Thread Profiler Help».
8. «Intel® Thread Profiler – краткое описание», материалы по образовательному комплексу «Технологии разработки параллельных программ».
9. «Эффективная многопоточность», Алексей Ширшов, RSDN Magazine #2-2003.