МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет прикладной математики, информатики и механики

Кафедра ERP-систем и бизнес процессов

Курсовая работа по теме:

Планирование распределенных вычислений.  
Широковещательный алгоритм Сузуки-Касами.

01.03.02. Прикладная математика и информатика

|  |  |  |  |
| --- | --- | --- | --- |
| Обучающийся |  |  | М.С.Поляков |
| Руководитель |  | ст.преп. | Ю.А.Крыжановская |

Воронеж 2018

**Содержание**

Введение3

Постановка задачи………………………………………………………………...3

1. Основные понятия и определения ………………………………………..4
2. Свойства распределенных систем
   1. Прозрачность………………………………………………………..5
   2. Открытость…………………………………………………………..7
   3. Масштабируемость………………………………………………….8
3. Модели распределенных систем
   1. Алгоритмы на основе передачи маркера…………………………..9
   2. Широковещательный алгоритм Сузуки-Касами…………………11
4. Реализация алгоритма………………………………………………………..14
   1. Структура приложения…………………………………………….16
   2. Работа приложения………………………………………………...22

Заключение24

Список литературы25

Приложение 126

**Введение**

В конце XX века компьютерные сети получили широкое распространение. Интерес к Information Technology (IT) стал проявляться в различных сферах жизнедеятельности человека таких как: экономической, социальной и других.

В связи с этим, постоянно растет объём данных для обработки. Несмотря на научно-технический прогресс, усовершенствований ЭВМ недостаточно для решения требуемых задач. Соответственно, возникает потребность в более оптимизированных алгоритмах и системах. Во всех современных продуктах программного обеспечение требуется параллельное и распределенное программирование.

Целью моей курсовой работы является изучение алгоритмов распределенных вычислений.

**Постановка задачи**

Входе работы были сформулированы следующие задачи:

1. Изучить виды и принципы построения распределенных систем
2. Смоделировать работу широковещательного алгоритма Сузуки-Касами

**1. Основные понятия и определения**

В литературе можно найти различные определения распределенных систем, причем ни одно из них не является удовлетворительным и не согласуется с остальными.

Распределенная система — это набор независимых компьютеров, представляющийся их пользователям единой объединенной системой.

В этом определении оговариваются два момента. Первый относится к аппаратуре: все машины автономны. Второй касается программного обеспечения: пользователи думают, что имеют дело с единой системой.

Также распределенные вычисления иногда определяют в более узком смысле, как применение распределенных систем для решения трудоемких вычислительных задач. В таком контексте распределенные вычисления являются частным случаем параллельных вычислений.

Отметим, что при изучении параллельных вычислений основной акцент обычно делается на методах разделения решаемой задачи на подзадачи, которые могут рассчитываться одновременно для максимального ускорения вычислений. Основная же особенность в организации параллельных вычислений с использованием распределенных систем будет заключаться в необходимости учитывать различие характеристик доступных вычислительных устройств и наличие существенной временной задержки при обмене данными между ними.

**2. Свойства распределенных систем**

**2.1. Прозрачность**

Важная задача распределенных систем состоит в том, чтобы скрыть тот факт, что процессы и ресурсы физически распределены по множеству компьютеров. Распределенные системы, которые представляются пользователям и приложениям в виде единой компьютерной системы, называются прозрачными.

Прозрачность доступа призвана скрыть разницу в представлении данных и в способах доступа пользователя к ресурсам. Так, при пересылке целого числа с рабочей станции на базе процессора Intel на Sun Scalable Processor ARChitecture (SPARC) необходимо принять во внимание, что процессоры Intel оперируют с числами формата «младший — последним» (первым передается старший байт), а процессор SPARC использует формат «старший — последним» (первым передается младший байт). Также в данных могут присутствовать и другие несоответствия. Например, распределенная система может содержать компьютеры с различными операционными системами, каждая из которых имеет собственные ограничения на способ представления имен файлов. Разница в ограничениях на способ представления имен файлов, так же как и собственно работа с ними, должны быть скрыты от пользователей и приложений[4].

Прозрачность местоположения призвана скрыть от пользователя, где именно физически расположен в системе нужный ему ресурс. Важную роль в реализации прозрачности местоположения играет именование. Так, прозрачность местоположения может быть достигнута путем присвоения ресурсам только логических имен, то есть таких имен, в которых не содержится закодированных сведений о местоположении ресурса. Примером такого имени может быть Uniform Resource Locator (URL): http://wiv.prenhall.com/index.html, в котором не содержится никакой информации о реальном местоположении главного web-сервера издательства Prentice Hall. О распределенных системах, в которых смена местоположения ресурсов не влияет на доступ к ним, говорят как об обеспечивающих прозрачность переноса. Более серьезна ситуация, когда местоположение ресурсов может измениться в процессе их использования, причем пользователь или приложение ничего не заметят. В этом случае говорят, что система поддерживает прозрачность смены местоположения.

Так, ресурсы могут быть реплицированы для их лучшей доступности или повышения их производительности путем помещения копии неподалеку от того места, из которого к ней осуществляется доступ. Прозрачность репликации позволяет скрыть тот факт, что существует несколько копий ресурса. Соответственно, система, которая поддерживает прозрачность репликации, должна поддерживать и прозрачность местоположения, поскольку иначе невозможно будет обращаться к репликам без указания их истинного местоположения.

Во многих случаях совместное использование ресурсов достигается посредством кооперации, например в случае коммуникаций. Однако существует множество примеров настоящего совместного использования ресурсов. Например, два независимых пользователя могут сохранять свои файлы на одном файловом сервере или работать с одной и той же таблицей

в совместно используемой базе данных. Это явление называется прозрачностью параллельного доступа. Отметим, что подобный параллельный

доступ к совместно используемому ресурсу сохраняет этот ресурс в непротиворечивом состоянии. Непротиворечивость может быть обеспечена механизмом блокировок, когда пользователи, каждый по очереди, получают исключительные права на запрашиваемый ресурс[2].

Последний тип прозрачности, который обычно ассоциируется с распределенными системами, — это прозрачность сохранности*,* маскирующая реальную или виртуальную сохранность ресурсов. Так, например, многие объектно-ориентированные базы данных предоставляют возможность непосредственного вызова методов для сохраненных объектов.

**2.2. Открытость**

Открытая распределенная система — это система, предлагающая службы, вызов которых требует стандартные синтаксис и семантику.

В распределенных системах службы обычно определяются через интерфейсы, которые часто описываются при помощи языка определения интерфейсов. Наиболее сложно точно описать то, что делает эта служба, то есть семантику интерфейсов. На практике подобные спецификации задаются неформально, посредством естественного языка.

Будучи правильно описанным, определение интерфейса допускает возможность совместной работы произвольного процесса, нуждающегося в таком интерфейсе, с другим произвольным процессом, предоставляющим этот интерфейс. Определение интерфейса также позволяет двум независимым группам создать абсолютно разные реализации этого интерфейса для двух различных распределенных систем, которые будут работать абсолютно одинаково. Правильное определение самодостаточно и нейтрально. «Самодостаточно» означает, что в нем имеется все необходимое для реализации интерфейса. Однако многие определения интерфейсов сделаны самодостаточными не до конца, поскольку разработчикам необходимо включать в них специфические детали реализации.

Следующая важная характеристика открытых распределенных систем — это гибкость. Под гибкостью мы понимаем легкость конфигурирования системы, состоящей из различных компонентов, возможно от разных производителей. Не должны вызывать затруднений добавление к системе новых компонентов или замена существующих, при этом прочие компоненты, с которыми не производилось никаких действий, должны оставаться неизменными. Другими словами, открытая распределенная система должна быть расширяемой. Например, к гибкой системе должно быть относительно несложно добавить части, работающие под управлением другой операционной системы, или даже заменить всю файловую систему целиком[1].

**2.3. Масштабируемость**

Масштабируемость системы может измеряться по трем различным показателям. Во-первых, система может быть масштабируемой по отношению к ее размеру, что означает легкость подключения к ней дополнительных пользователей и ресурсов.

Во-вторых, система может масштабироваться географически, то есть пользователи и ресурсы могут быть разнесены в пространстве.

В-третьих система может быть масштабируемой в административном смысле, то есть быть проста в управлении при работе во множестве административно независимых организаций. К сожалению, система, обладающая масштабируемостью по одному или нескольким из этих параметров, при масштабировании часто дает потерю производительности.

Поскольку проблемы масштабируемости в распределенных системах, такие как проблемы производительности, вызываются ограниченной мощностью серверов и сетей, существуют три основные технологии масштабирования: сокрытие времени ожидания связи, распределение и репликация[3].

Сокрытие времени ожидания связи применяется в случае географического масштабирования. Основная идея проста: постараться по возможности избежать ожидания ответа на запрос от удаленного сервера. Например, если была запрошена служба удаленной машины, альтернативой ожиданию ответа от сервера будет осуществление на запрашивающей стороне других возможных действий.

В сущности, это означает разработку запрашивающего приложения в расчете на использование исключительно асинхронной связи. Когда будет получен ответ, приложение прервет свою работу и вызовет специальный обработчик для завершения отправленного ранее запроса. Для осуществления запроса может быть запущен новый управляющий поток выполнения. Хотя он будет блокирован на время ожидания ответа, другие потоки процесса продолжат свое выполнение.

**3.Модели распределенных систем**

**3.1. Алгоритмы на основе передачи маркера**

Для алгоритмов данного класса право войти в компьютерную сеть (КС) материализуется в виде уникального объекта – маркера, который в каждый момент времени может содержаться только у одного процесса или же находиться в канале в состоянии пересылки от одного процесса к другому. Свойство безопасности алгоритмов взаимного исключения в этом случае будет гарантировано ввиду уникальности маркера. При этом процесс, владеющий маркером, может неоднократно входить в КС до тех пор, пока маркер не будет передан другому процессу. Очевидно, что в течение всего времени выполнения внутри КС процесс должен удерживать маркер у себя.

Главные различия алгоритмов, основывающихся на передаче маркера, заключаются в методах поиска и получения маркера, причем эти методы должны гарантировать, что маркер рано или поздно окажется в каждом процессе, желающем войти в КС. Самым простым решением, обеспечивающим такие гарантии, является организация непрерывного перемещения маркера среди всех процессов распределенной системы. Чтобы не пропустить ни одного процесса, желающего войти в КС, все процессы распределенной системы обычно организуют в направленное логическое кольцо, по которому и циркулирует маркер.

Алгоритмы с таким механизмом перемещения маркера называют алгоритмами маркерного кольца. Любой процесс, желающий войти в КС, ожидает прихода маркера, и после выхода из КС передает его дальше по кольцу. Если процесс, получивший маркер, не заинтересован в работе с КС, он просто передает маркер своему соседу. Поэтому даже если ни один из процессов не работает с КС, маркер продолжает непрерывно циркулировать между процессами. Поскольку маркер перемещается от процесса к процессу в общеизвестном порядке, ситуации голодания возникнуть не может[1]. Когда процесс решает войти в КС, в худшем случае ему придется ожидать, пока все остальные процессы последовательно войдут в КС и выйдут из нее в виде уникального объекта – маркера, который в каждый момент времени может содержаться только у одного процесса или же находиться в канале в состоянии пересылки от одного процесса к другому. Свойство безопасности алгоритмов взаимного исключения в этом случае будет гарантировано ввиду уникальности маркера. При этом процесс, владеющий маркером, может неоднократно входить в КС до тех пор, пока маркер не будет передан другому процессу. Очевидно, что в течение всего времени выполнения внутри КС процесс должен удерживать маркер у себя[2].

**3.2.Широковещательный алгоритм Сузуки-Касами**

Одним из решений, позволяющих каждому запросу на владение маркером гарантированно достичь процесса, в котором находится маркер, является широковещательная рассылка таких запросов всем процессам распределенной системы. Данный алгоритм был предложен И. Сузуки и Т. Касами, и суть его заключается в следующем. Если процесс, не обладающий маркером, собирается войти в КС, он рассылает всем другим процессам сообщение REQUEST с запросом на владение маркера. При получении сообщения REQUEST процесс, владеющий маркером, пересылает его запрашивающему процессу. Если в момент получения сообщения REQUEST процесс, владеющий маркером, выполняется внутри КС, он откладывает передачу маркера до тех пор, пока не выйдет из нее.

Несмотря на кажущуюся простоту описанной схемы взаимодействия, представленный алгоритм передачи маркера должен эффективно справляться с решением двух связанных между собой задач[4].

1. Необходимо различать устаревшие запросы на получение маркера от текущих, еще необслуженных запросов. Действительно, запрос на владение маркером получают все процессы, однако, удовлетворить этот запрос может только один процесс, владеющий маркером. В результате, после того как запрос будет обслужен, у остальных процессов будут находиться устаревшие запросы, в ответ на которые не нужно передавать маркер. В случае, если процесс не имеет возможности определить, был ли находящийся у него запрос уже обслужен или нет, он может передать маркер процессу, который на самом деле в нем не нуждается. Это не приведет к нарушению корректности работы алгоритма взаимного исключения, но может серьезно сказаться на производительности системы.
2. Необходимо вести перечень процессов, ожидающих получения маркера. После завершения процессом своего выполнения внутри КС, он должен определить список процессов, находящихся в состоянии запроса на вход в КС, для того, чтобы передать маркер одному из них.

Для решения этих задач в алгоритме Сузуки-Касами каждый процесс использует порядковые номера запросов на вход в КС. Порядковый номер n (n = 1, 2, …) увеличивается процессом Pi независимо от других процессов каждый раз, когда Pi формирует новый запрос на вход в КС. Порядковый номер запроса Pi передается в рассылаемом сообщении REQUEST в виде REQUEST(i, n). Кроме того, каждый процесс Pi поддерживает работу с массивом Rni[1..N], где в элементе Rni[j] содержится наибольшее значение порядкового номера запроса, полученного от процесса Pj. Другими словами, при получении сообщения REQUEST(j, n) процесс Pi изменяет значение j-го элемента своего массива Rni согласно выражению Rni[j] = max(Rni[j], n).[4]

Таким образом, алгоритм Сузуки-Касами будет определяться следующими правилами.

1. Запрос на вход в КС.

• Если желающий войти в КС процесс Рi не владеет маркером, он увеличивает на единицу порядковый номер своих запросов Rni[i] и рассылает всем другим процессам сообщение REQUEST(i, n), где n – обновленное значение Rni[i].

• Когда процесс Рj получает запрос REQUEST(i, n) от процесса Рi, он изменяет значение элемента массива RNj[i] = max(RNj[i], n). Если при этом Рj владеет маркером и находится в состоянии выполнения вне КС, то Рj передает маркер процессу Рi при условии, что RNj[i] = LN[i] + 1.

1. Вход в КС. Процесс Рi может войти в КС, если он обладает

маркером.

1. Выход из КС.

• При выходе из КС процесс Рi обновляет в маркере значение элемента массива LN[i]: LN[i] = Rni[i].

• Для каждого процесса Pj, идентификатор которого не присутствует в очереди Q, если Rni[j] = LN[j] + 1, то процесс Рi добавляет идентификатор Pj в конец очереди Q.

• Если после выполнения представленных выше операций с очередью Q она оказывается не пустой, процесс Рi выбирает первый идентификатор из Q (при этом удаляя его из очереди) и передает маркер процессу с этим идентификатором.

Покажем, что алгоритм Сузуки-Касами обладает свойством живучести. Действительно, запрос процесса Рi на вход в КС за конечное время достигнет других процессов в системе. Один из этих процессов обладает маркером (или, в конце концов, получит его, если маркер находится в состоянии пересылки между процессами). Поэтому запрос процесса Рi рано или поздно будет помещен в очередь Q. Перед ним в очереди может оказаться не более (N – 1) запросов от других процессов, и, следовательно, в конце концов, запрос от Рi будет обслужен, и Рi получит маркер[4].

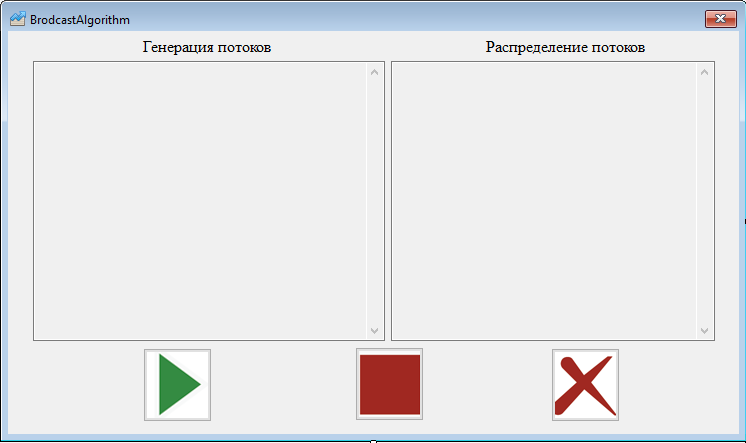
В заключение отметим, что если процесс не владеет маркером, то для входа в КС алгоритм Сузуки-Касами требует обмен N сообщениями: (N – 1) сообщений REQUEST плюс одно сообщение для передачи маркера. Если же процесс владеет маркером, то для входа в КС ему не потребуется ни одного сообщения.

**4.Реализация алгоритма**

Для реализации и моделирования работы широковещательного алгоритма Сузуки-Касами будем использовать среду разработки Microsoft Visual Studio 2017. В Visual Studio выберем платформу .NET Framework в которую непосредственно включён язык программирования C#.

В качестве КС используем буферы потоков. Каждый буфер моделирует работу одного компьютера в сети. Роль REQUEST запросов от компьютера к серверу будут играть множество потоков внутри буфера. Приложение соответственно является сервером, обрабатывающем потоки.

Поток содержит в себе информацию о своём приоритете, аналогично запросам в КС. Однако, поскольку все происходит в рамках одного компьютера, маркер не обязан содержать в себе массив приоритетов запросов, потому, что он уже содержится в буфере. Поэтому маркером, который передают потоки будет служить стандартная булевская переменная.

Рис.1. - AppForm

При создании проекта выберем приложение Windows Forms. Для моделирования достаточно одной формы с некоторыми элементами.

Тексты “Генерация потоков” и “Распределение потоков” разместим в элементах Label. Они помогут пользователю быстрее понять работу приложения

Ниже создадим два TextBox элемента в которых будем выводить информацию о работе алгоритма. Под двумя TextBox расположим три интерактивных объекта Button которые будут выполнять действия, соответствующие своим иконкам – “Пуск”, “Стоп”, “Выход”. На этом интерфейс приложения можно считать законченным.

**4.1.Структура приложения**

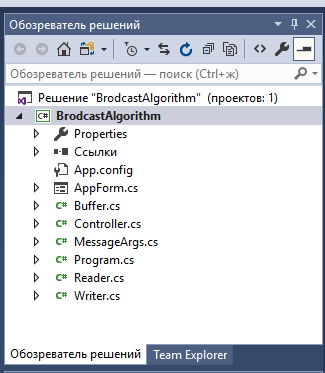


Рис.2. - Файлы проекта

Опишем все элементы структуры приложения. AppForm.cs – это единственная форма, интерфейс которой описан выше.

Класс Buffer.cs внутри себя содержит поле с именем типа String. Поле очередь типа Queue в которой хранятся приоритеты потоков. А также три поля типа Semaphore, которые призваны регулировать количество потоков, получающих доступ к одним и тем же ресурсам.

В методах класса описан один конструктор, для создания объекта по умолчанию. Методы public void WriteMessage(int x) и public void ReadMessage() предназначены для записи и чтения элементов класса соответственно.

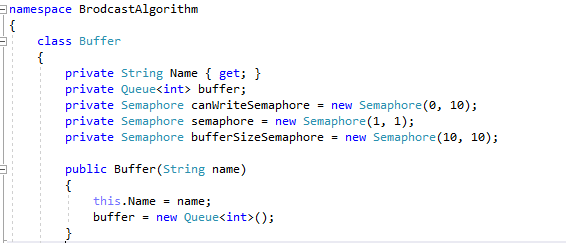


Рис.3. – Класс Buffer

Класс Controller.cs основной элемент управления приложением выполняющий функции генерации поток. Поле form типа AppForm содержит единственную форму для удобного взаимодействия с интерфейсом приложения.

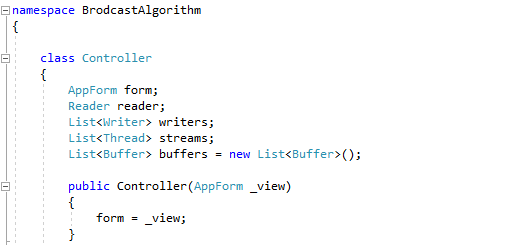


Рис.4. – Класс Controller

Поля reader и список writers хранят информацию о записанных и считанных потоках. Streams типа List<Thread> это список потоков сгенерированных приложением. В списки буферов находят все сгенерированные объекты класса Buffer описанного выше.

Внутри класса конструктор по умолчанию связывает форму с контроллером. Метод public void InitializeController() инициализирует первый буфер при запуске приложения, а так же добавляет в него несколько потоков. Вызываются методы Writer.cs для вывода информации в окна TextBox.

public void AddWriter() – создает новые элементы класса Buffer уже после инициализации первого, для длительной работы программы.

Методы : public void Stop(), public void Start(),public void StopController() призваны реализовать интерактивные кнопки на форме приложения. Public void Stop() – вызывает остановку всех действующих потоков в том числе и генерацию новых. Public void Start() – возобновляет действия потоков, после остановки их пользователем. Public void StopController() – выполняет завершения всех действующих потоков перед закрытием приложения.

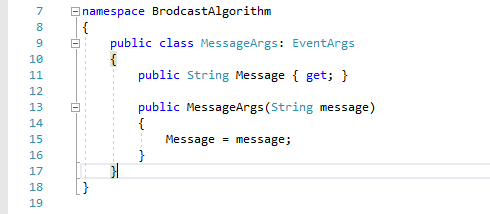


Рис.5. – Класс MessageArgs

MessageArgs.cs небольшой класс с всего лишь одним String полем и конструктором. Он наследован от базового C# класса который содержит данные событий, и предоставляет значения для событий, не содержащих данные.

Program.cs файл автоматически сгенерированный Visual Studio. Он выполняет запуск программы, отрисовывает форму с её интерфейсом.

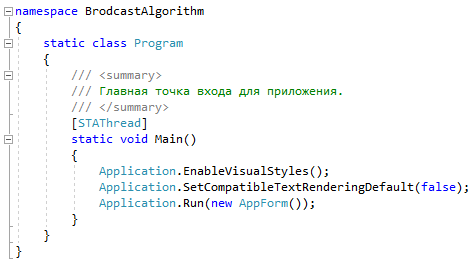


Рис.6. – файл Program.cs

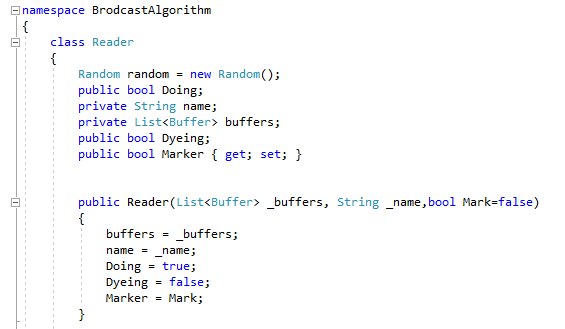


Рис.7. – Класс Reader.cs

Класс Reader реализует чтение буфера с потоками, которые генерирует контроллер приложения.

Опишем поля Reader.cs : переменная random типа Random требуется для моделирования приоритетности потока, которую мы будем задавать случайным образом. Bool Doing флаг определяет, находятся ли в данном буфере потоки, которые выполняются. Поле bool Dyeing сигнализирует о завершении всех возможных потоков внутри буфера.

String name дублирует поле внутри класс Buffer и хранит в себе имя переданного объекта этого класса. Список List<Buffer> содержит все буферы которые были считаны после генерации их контроллером. Логическая переменная bool Marker информирует о наличии маркера для возможности входа в смоделированную КС. Конструктор класса получает данные о буфере и его имени, все остальные поля заполняются по умолчанию.

Метод public void ReadMessage() обеспечивает чтение потока владеющего в данным момент маркером, если это невозможно, то считывает случайный поток из буфера, что не противоречит алгоритму Сузуки-Касами.

Public void Work() метод вызываемый контроллером, для запуска еще “живых” буферов, внутри него уже непосредственно используется ReadMessage(). Функции Pause(), Resume(), Die() реализуют команды контроллера, которые он в свою очередь получает от пользователя посредством события - нажатия на кнопки основной формы AppForm.

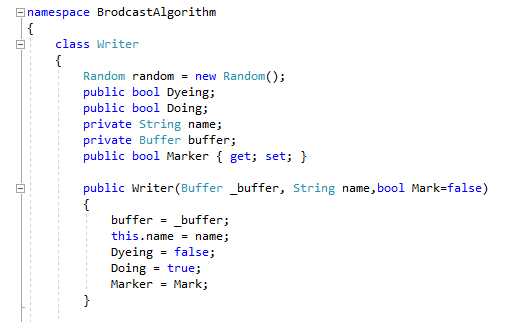


Рис.8. – Writer.cs

Класс Writer очень похож на Reader потому, что выполняет те же функции, за исключением того, что выполняет не чтение, а запись данных на форму. Buffer buffer содержит внутри себя текущий буфер, готовящийся к записи. String name несет в себе имя этого буфера.

Логические переменные Doing и Dyeing определяют обрабатываются ли потоки, и существуют ли еще не обработанные. Marker дублирует значение присвоенное в классе Reader. Конструктор полностью аналогичен предыдущему описанному выше. Public void PrintMessage() если буфер не пуст, печатает сообщение о выполнение потока с его приоритетностью, если поток оказался уже завершенным, то указывается приоритетность -1.

Метод Work() вызывается контроллером и внутри содержит PrintMessage(), пока выполнены условия: буфер еще работает и есть хотя бы один не “умерший” поток. Функции Pause(), Resume(), Die() реализуют команды контроллера поступающие от интерфейса приложения : остановка потоков, продолжение работы потоков, завершение приложения. Для этого они меняют соответствующие логические переменные класса.

**4.2.Работа приложения**

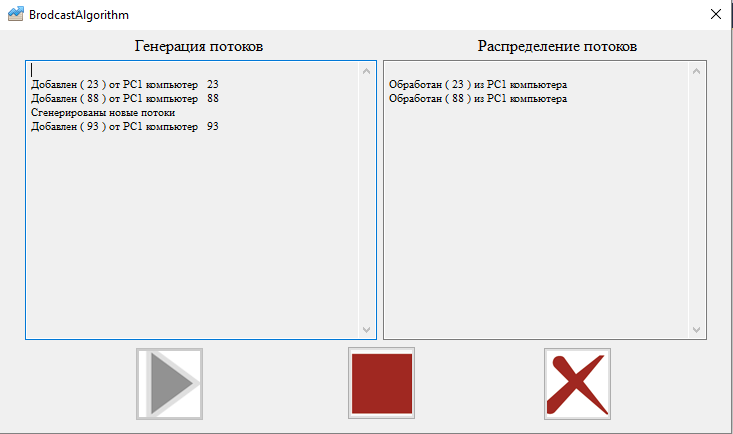


Рис.10. – Начало работы

В левом окне “Генерация потоков” создан первый (компьютер) буфер, который начал отправлять (запросы) потоки данному (серверу) приложению. Поскольку потоков еще немного, быстродействие приложения позволяет обрабатывать их в порядке появления, приоритет не учитывается. Потоки, которые были обработаны и завершены, выведены на правое окно “Распределение потоков”.

Однако, при увеличении количества буферов и, соответственно количества потоков, простого быстродействия недостаточно. В этом месте поможет распределенная система с алгоритмом Сузуки-Касами. Рассмотрим подробнее на рисунке ниже.

На рисунке видно, что количество компьютеров в сети (буферов) отправляющих запросы (потоки) увеличилось. Ресурсов сервера (приложения) недостаточно, что успевать сразу обрабатывать все запросы в порядке их поступления.

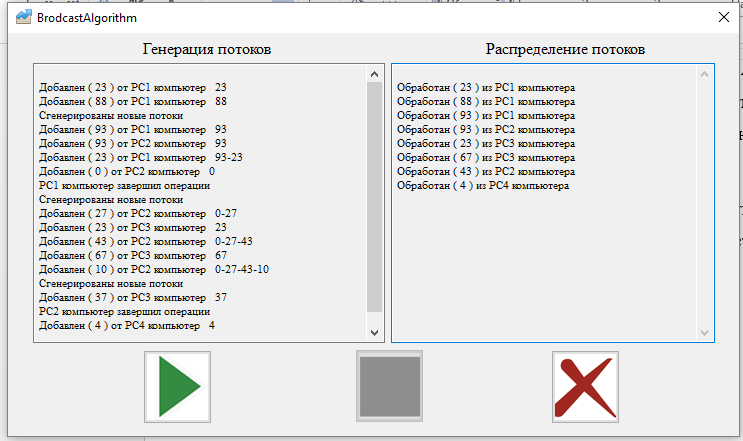


Рис.11. – Работа приложения

После обработки запросов первого буфера (PC1), алгоритм передает маркер самому приоритетному запросу от PC2. После его завершения, программа передала маркер случайному запросу так как, в момент передачи не было непосредственно произведён запрос. Затем приложения обработает два запроса в порядке их приоритета, передавая им маркер. Далее при передаче маркера, поступил запрос от PC4, который был сразу обработан. Следующая передача будет снова осуществляться по приоритету.

Для завершения приложения достаточно нажать кнопку “Выход”.

**Заключение**

В результате работы были изучены основные понятия распределенных систем и их важнейшие свойства. Подробно рассмотрен один из множества алгоритмов распределенных систем, который оптимизируют работу КС. Смоделирована его работа, путем написания многопоточного оконного приложения.

**Список литературы**

1. Таненбаум Э., ван Стеен М. Распределенные системы. Принципы и парадигмы. – СПб: Питер, 2003. – 877 с.: ил.
2. Тель Ж. Введение в распределенные алгоритмы. Пер. с англ. – М.: МЦНМО, 2009. – 616 с.
3. Топорков В. В. Модели распределенных вычислений. – М.: Физматлит, 2004. – 320 с.
4. Косяков М. С. Введение в распределенные вычисления. – СПб: НИУ ИТМО,2014. – 155с.
5. Булгакова И.Н. Методические рекомендации по разработке, оформлению и защите курсовых и выпускных квалификационных работ. – Воронеж: ВГУ,2016. – 31c.

**Приложение 1**

class Buffer

{

private String Name { get; }

private Queue<int> buffer;

private Semaphore canWriteSemaphore = new Semaphore(0, 10);

private Semaphore semaphore = new Semaphore(1, 1);

private Semaphore bufferSizeSemaphore = new Semaphore(10, 10);

public Buffer(String name)

{

this.Name = name;

buffer = new Queue<int>();

}

public event EventHandler<MessageArgs> OnMessageAdd;

public event EventHandler<MessageArgs> OnMessageRemove;

/// <summary>

/// Добавление элемента в буфер

/// </summary>

/// <param name="x">Элемент для добавления</param>

public void WriteMessage(int x)

{

bufferSizeSemaphore.WaitOne();

semaphore.WaitOne();

buffer.Enqueue(x);

OnMessageAdd?.Invoke(this, new MessageArgs(x > -1 ? "Добавлен ( " + x + " ) от " + Name + " " + string.Join("-", buffer.ToArray()) : Name + " завершил операции"));

semaphore.Release();

canWriteSemaphore.Release();

}

/// <summary>

/// Чтение элемента из буфера

/// </summary>

public void ReadMessage()

{

canWriteSemaphore.WaitOne();

semaphore.WaitOne();

OnMessageRemove?.Invoke(this, new MessageArgs("Обработан ( " + buffer.Dequeue() + " ) из " + Name + "а"));

semaphore.Release();

bufferSizeSemaphore.Release();

}

}

}

class Controller

{

AppForm form;

Reader reader;

List<Writer> writers;

List<Thread> streams;

List<Buffer> buffers = new List<Buffer>();

public Controller(AppForm \_view)

{

form = \_view;

}

/// <summary>

/// Связывает контроллер и добаляет первый поток

/// </summary>

public void InitializeController()

{

Buffer bufferW1 = new Buffer("PС1 компьютер");

buffers.Add(bufferW1);

bufferW1.OnMessageAdd += form.OnMessageAddToBuffer;

bufferW1.OnMessageRemove += form.OnMessageRemoveFromBuffer;

reader = new Reader(buffers, "Reader");

Writer writer = new Writer(bufferW1, "Writer1",true);

writers = new List<Writer> { writer };

streams = new List<Thread> { new Thread(reader.Work), new Thread(writer.Work) };

foreach (var t in streams) t.Start();

}

/// <summary>

/// Добавление новый поток

/// </summary>

public void AddWriter()

{

Buffer buf = new Buffer("PC" + (writers.Count + 1) + " компьютер");

buffers.Add(buf);

buf.OnMessageAdd += form.OnMessageAddToBuffer;

buf.OnMessageRemove += form.OnMessageRemoveFromBuffer;

Writer writer = new Writer(buf, "Writer" + (writers.Count + 1));

writers.Add(writer);

Thread th = new Thread(writer.Work);

streams.Add(th);

th.Start();

}

public void Stop()

{

reader.Pause();

foreach (var w in writers) w.Doing = false;

}

public void Start()

{

reader.Resume();

foreach (var w in writers) w.Doing = true;

}

public void StopController()

{

reader.Die();

foreach (Writer writer in writers)

{

writer.Die();

}

foreach (Thread thread in streams)

{

thread.Abort();

}

}

}

}

class Reader

{

Random random = new Random();

public bool Doing;

private String name;

private List<Buffer> buffers;

public bool Dyeing;

public bool Marker { get; set; }

public Reader(List<Buffer> \_buffers, String \_name,bool Mark=false)

{

buffers = \_buffers;

name = \_name;

Doing = true;

Dyeing = false;

Marker = Mark;

}

public void AddWriter(Buffer \_buffer)

{

buffers.Add(\_buffer);

}

/// <summary>

/// </summary>

public void ReadMessage()

{

if (buffers != null && Marker)

{

int max = buffers.ElementAt(0);

foreach (var w in buffers)

if (buffers.ElementAt(w)>max) max=buffers.ElementAt(w);

var buffer = buffers.ElementAt(max);

buffer.ReadMessage();

}

else

{

var buffer = buffers.ElementAt(random.Next(0, buffers.Count));

buffer.ReadMessage();

}

Thread.Sleep(3000);

}

public void Work()

{

while (!Dyeing)

{

while (Doing && !Dyeing)

{

ReadMessage();

}

}

}

public void Pause()

{

Doing = false;

}

public void Resume()

{

Doing = true;

}

public void Die()

{

Dyeing = true;

Marker = false;

}

}

}

class Writer

{

Random random = new Random();

public bool Dyeing;

public bool Doing;

private String name;

private Buffer buffer;

public bool Marker { get; set; }

public Writer(Buffer \_buffer, String name,bool Mark=false)

{

buffer = \_buffer;

this.name = name;

Dyeing = false;

Doing = true;

Marker = Mark;

}

public void PrintMessage()

{

if (buffer != null)

{

int value = random.Next(0, 106);

Thread.Sleep(3000);

if (value <= 100)

buffer.WriteMessage(value);

else

{

buffer.WriteMessage(-1);

Die();}}}

public void Work()

{

while (!Dyeing)

{

while (Doing && !Dyeing)

{

PrintMessage();

}}}

public void Die()

{

Dyeing = true;

Marker = false; }

public void Pause()

{

Doing = false;}

public void Resume()

{Doing = true;}}}