СПОСОБЫ ПОВЫШЕНИЯ ОТЗЫВЧИВОСТИ СЕРВЕРА НА ПРИМЕРЕ РАЗРАБОТКИ ОНЛАЙН СЕРВИСА ДЛЯ ПРОВЕДЕНИЯ ИНТЕЛЛЕКТУАЛЬНЫХ ИГР

Рогазинский А.А., студент кафедры АОИ

Научный руководитель: Сидоров А.А., канд. экон. наук, доцент кафедры АОИ

За последнее время требования к программам значительно изменились. Ещё несколько лет назад веб-сервисы могли позволить себе секунды ожидания ответа, часы простоя из-за необходимой доработки и гигабайты данных. Сегодня, пользователь требует мгновенного ответа и бесперебойной работы программной системы, а объёмы информации, обрабатываемой приложениями, измеряются в петабайтах. В итоге, возникают случаи, когда используемые подходы к реализации программной архитектуры не позволяют программам удовлетворять всем требованиям пользователей [1].

В рамках проекта “Интерактивные сервисы” разрабатываются программные системы предоставляющие пользователям возможности удаленного взаимодействия с использованием мобильных устройств. Одна из главных задач при этом сохранить отзывчивость сервиса даже при высоких нагрузках. При этом необходимо учитывать тот факт, что при разработке клиентской части функциональные возможности во многом зависят от пользователя и технических характеристик его смартфона. В то же время ответственность за функциональные возможности сервера целиком ложится на разработчиков. Суть проекта “Интерактивные сервисы” заключается в формировании среды, позволяющей множеству пользователей одновременно обращаться к определенному веб-сервису. С практической точки зрения это может быть использовано, например, при организации интерактивных интеллектуальных игр в режиме онлайн.

Чаще всего выделяют следующие проблемы, когда речь заходит об одновременной обработке большого количества клиентов [2]:

- Проблема ввода-вывода, связанная с ситуацией, когда время, необходимое для обработки запроса, значительно зависит от времени, потраченного на выполнение операций ввода-вывода;
- Неоптимальное использование многоядерных процессоров и возможностей операционной системы.

Для решения данных проблем обычно рекомендуют применять следующие подходы при разработке:

- Использование многозадачности в каком-либо виде – стандартный подход при разработке нагруженной программной системы, особенно в нынешнюю эпоху интернета;
- Использование неблокирующих вызовов для операций ввода-вывода по сети, а также уведомления об их завершении. Для этого существуют специальные сервисы уведомлений, предоставляемые операционными системами, например, *poll, kqueue* [3].

Многозадачность позволяет обеспечить параллельную обработку нескольких запросов пользователей, в некоторых случаях даже с оптимальным использованием вычислительных ресурсов сервера. Её принято разделять на два вида: *невытесняющую*(*non-preemptive*) и *вытесняющую* многозадачность (*preemptive multitasking*). В системах с *невытесняющей* многозадачностью работа любого процесса может быть прервана только «по инициативе самого процесса», а точнее – только когда процесс вызывает определенные системные функции. Может также иметься функция, специально предназначенная для добровольной уступки процессом очереди на выполнение *(Yield)*. Первые операционные системы Windows реализовывали именно этот тип многозадачности. При этом программы для данных ОС легко могли вызвать зависание всей системы при неправильной разработке. В настоящее время в языках программирования такой вид многозадачности часто

представлен в виде возможности создавать *сопрограммы(coroutines)*, управляемые встроенным в программы планировщиком.

Особенность вытесняющей многозадачности заключается в том, что планировщик задач вступает в работу не только при вызове системных функций, но и когда активизируется процесс, обладающий более высоким приоритетом, чем текущий, либо когда истекает квант времени, выделенный планировщиком для текущего процесса. Большинство современных операционных систем используют именно вытесняющую многозадачность для параллельного выполнения нескольких процессов. В языках программирования вытесняющая многозадачность обычно представлена в виде возможности создавать потоки(*threads*), выполняющиеся параллельно на различных ядрах процессора, но в то же время имеющие общую область памяти. Наиболее популярны следующие способы добиться многозадачности в программной системе [4]:

– Системные потоки (*threads*);
– Сопрограммы (*Coroutines*).

Первый способ, самый известный и самый сложный – системные потоки (*threads*). Преимущества данного подхода в высокой скорости обработки и оптимальном использовании многоядерного процессора. В то же время, написать многопоточную программу намного сложнее, чем, к примеру, событийно-ориентированную. Данный способ рекомендуется только для программистов, имеющих опыт работы с подобными проектами [5, 6].Рассмотрим причину этой сложности.

При анализе части обычной выполняющейся последовательно программы, изменяющей значения каких-либо глобальных переменных, достаточно отследить состояние всей системы перед и в момент последовательного выполнения её команд. Таким образом, если эта часть состоит из n инструкций, то количество уникальных ситуаций для рассмотрения — $n + 1$. С другой стороны, в той же части программы в случае многопоточного её выполнения число уникальных ситуаций изменения состояния системы, которые необходимо отследить увеличивается до n^m, где m — количество потоков в системе. Это пример сложности разработки программы при использовании системных потоков.

Также, сейчас возможно разрабатывать программные системы, содержащие множество компонентов. Однако, в связи с особенностями человеческого мозга необходимо иметь возможность вносить изменения в эту систему одновременно учитывая примерно семь каких-либо фактов об этой системе [7]. В случае многопоточной программы это вызывает проблемы. Также, при разработке программ с использованием такого языка, как например, Python, можно столкнуться с тем фактом, что все поля и методы объектов являются открытыми и могут быть изменены извне. При разработке многопоточной программы значительно проще сделать ошибку в одном месте, которая нарушит работу всей системы, при этом даже не осознавая это. Приходится придерживаться высокого уровня бдительности, чтобы убедиться что все соглашения о том, кто и когда может изменять значения полей объектов, соблюдены. Иначе будет практически невозможно отследить источник проблемы.

Следующий способ – сопрограммы – менее известен, но тем не менее тоже находит применение в решении определенного круга задач. Сопрограммы – синтаксические конструкции языка, позволяющие явно приостановить действие сопрограммы (подзадачи) в определённом месте и переключится на выполнение другой. Примеры можно найти в таких языках как Python(*@inlineCallback, yield from*), C#(*async/await*). Преимущества использования этого подхода в упрощении написания и чтения кода, способного обрабатывать несколько запросов параллельно, легкой отладке программы и отсутствии ситуаций взаимных блокировок (одна из частых проблем многопоточных программ). Недостаток подхода – относительно относительно низкая производительность, в частности в языке Python. На практике переключения между сопрограммами часто используются для опти-

мизации работы в системе, где наблюдается проблемы длительного ввода/вывода [8]

Интересный факт, что некоторые языки программирования значительно продвинулись в реализации невытесняющей многозадачности. Так например Erlang, поддерживает работу с мини-процессами с изолированным состоянием и встроенным быстрым каналом для передачи неизменяемых значений. Помимо этого стоит отметить новый язык программирования Go, в котором разработчики языка особенно постарались над реализацией идеи сопрограмм. Так, в процессе работы программы встроенный runtime-планировщик Go переключается между несколькими «горутинами» в пределах одного потока операционной системы. Совершенно нормальная ситуация, когда в одном процессе выполняются десятки тысяч «горутин», не редкость и сотни тысяч. С точки зрения языка планирование выглядит как вызов функции и имеет ту же семантику. Компилятору известно, какие регистры процессора используются, и он автоматически их сохраняет. Сравните это с многопоточными приложениями, когда поток может быть вытеснен в любое время, в момент выполнения любой инструкции. Также, используется фиксированный набор ("пул") потоков программы и среда выполнения Go сама заботится о том, чтобы назначать готовые к исполнению «горутины» в свободные потоки.

Как было сказано раньше, помимо использования многозадачности существует другой способ оптимизации нагруженного приложения — использование неблокирующих (асинхронных) вызовов ввода и вывода. Стоит отметить, что применение асинхронных вызовов при обращении к базам данных рекомендуется только при наличии большого количества сложных запросов, выполнение которых можно производить параллельно [9]. Современные языки программирования предоставляют следующие возможности для написания асинхронного кода:

- Явные функции обратного вызова
- Объекты наподобие *Promise*-ов
- *Async/await* команды

Явные функции обратного вызова – один из первых подходов, суть его в том, что для определённого события назначается функция которая будет вызвана при возникновении этого события. После её выполнения управление будет передано назад или главному циклу программы (*main loop*), который также выполняет другие функции обратного вызова. Пример – *on<foo>* подход в JavaScript. Преимущества данного подхода – удобство отладки, при чтении программист всегда сможет узнать в какой момент времени выполняется тот или иной участок кода. Недостатки – написание дополнительного кода, менее читабельный вид, также называемый "*callback hell*"[10].

Кроме того, возможен подход, в котором вызываемая функция создаёт объект для хранения значения, которое будет доступно позже (*Future*), возвращает его вызывающей функции, для того чтобы та назначила функцию обратного вызова на событие появления реального значения в этом объекте. Пример – *Promise* в JavaScript.

Так, в JavaScript единственный способ добиться «многозадачности» и отзывчивого пользовательского интерфейса при выполнении длительных вычислений — использование промисов(*Promises*) и асинхронное выполнение кода. Также, в основе среды выполнения javascript кода Node.js лежит именно идея асинхронного ввода-вывода .

Помимо этого, на данный момент существуют библиотеки, реализующие автоматическое переключение сопрограмм при выполнении операции ввода-вывода. Причём визуально это выглядит как написание программы с применением сопрограмм и использованием синтаксиса *"async/await"*. Недостаток данного подхода в том, что механизм переключения скрыт от программиста, из-за чего ему предоставляется меньше возможностей для настройки под свои нужды.

Подводя итог, можно дать следующие рекомендации по использованию описанных выше подходов:

- Использовать потоки только там где это действительно необходимо, например для сложных научных вычислений, при этом рекомендуется изолировать эту часть программы в многопоточное ядро программы. Желательно оставлять большую часть программы однопоточной.
- При необходимости желательно использовать событийно-ориентированный подход (в частности, в пользовательских интерфейсах и распределенных системах)
- При построении программы с использованием асинхронных вызовов, лучше всего, реализовать ядро с применением явных функций обратного вызова и при этом реализовывать высокоуровневую логику с использованием сопрограмм.
- Желательно как бы достигать компромисса между разными подходами, с учетом их преимуществ и недостатков. Идеальных методов решения проблем нагруженного приложения не существует.

В заключение стоит отметить, что всё сказанное выше было применено в рамках ГПО при написании серверной части программной системы для проведения интерактивных интеллектуальных игр в режиме онлайн. В частности, был использован фреймворк *"Asyncio"* [11] для организации асинхронного ввода-вывода, использованы команды "*async*" и "*await*" для организации сопрограмм, рассмотрен вариант использования библиотеки для объектно-реляционное отображения (*ORM*) "*peewee*" для реализации асинхронного взаимодействия с базой данных, также планируется внедрение библиотеки "*concurent.futures*" для оптимизации использования многоядерных процессоров.

Список литературы

1 Манифест "Реактивного" подхода в разработке, цель которого – повышение отзывчивости веб-сервиса [Электронный ресурс] – http://www.reactivemanifesto.org/

2 Описание проблемы "10K" и вариантов её решения [Электронный ресурс] – http://www.kegel.com/c10k.html#aio%22C10K%20article%22

3 Об системных утилитах epoll, kqueue и для чего они нужны [Электронный ресурс] – http://austingwalters.com/io-multiplexing/

4 Об особенностях вытесняющей и невытесняющей многозадачности [Электронный ресурс] – https://sqljunkieshare.com/2012/01/06/preemptive-vs-non-preemptive-and-multitasking-vs-multithreading/

5 О проблемах, связанных с разработкой многопоточных приложений [Электронный ресурс] – https://glyph.twistedmatrix.com/2014/02/unyielding.html

6 Иллюстративное описание проблем, связанных с написанием многопоточных программ [Электронный ресурс] – http://web.stanford.edu/ ouster/cgi-bin/papers/threads.pdf

7 Магическое число семь плюс-минус два. О некоторых пределах нашей способности перерабатывать информацию – Дж. А. Миллер – Электронный доступ : http://www.ebbinghaus.ru/wp-content/uploads/2010/02/Miller_564-580.pdf

8 Об сопрограммах и способах их применения в Python [Электронный ресурс] – http://www.informit.com/articles/article.aspx?p=2320938

9 Сравнение многопоточного и асинхронного подходов для выполнения запросов к базе данных [Электронный ресурс] – http://techspot.zzzeek.org/2015/02/15/asynchronous-python-and-databases/

10 О правильном использовании функций обратного вызова [Электронный ресурс] – http://transceptor.technology/blog/avoiding-callback-hell-in-python/

11 Документация к стандартной библиотеке языка Python, предназначенной для написания асинхронных программ [Электронный ресурс] – http://pythonhosted.org/aiopg/