

Erlang: Еще раз

Андрей Ушаков рассматривает организацию работы многозадачных приложений. Без взаимодействия тут не обойтись...



Наш
эксперт

Андрей Ушаков
активно приближает тот день, когда функциональные языки станут мейнстримом.

Взаимодействие между задачами или их синхронизация – пожалуй, наиболее важная (и «больная») тема, когда мы думаем об организации нашего приложения в виде нескольких одновременно выполняющихся задач. Понятно, что без взаимодействия задач не создать правильно функционирующее многозадачное приложение (за редкими исключениями). «Больная» же эта тема потому, что с ней связаны практически все ошибки и проблемы многозадачности. Вот об этом и поговорим.

Первый и, пожалуй, главный вопрос, требующий ответа – что такое синхронизация между задачами и зачем она нужна. Синхронизация между задачами – это всего лишь взаимодействие между задачами в многозадачной среде при помощи тех или иных средств. А вот зачем оно нужно, уже интереснее. Крайне редко удается разделить одну большую задачу на несколько абсолютно независимых задач, способных выполняться параллельно. Чаще одни задачи должны дождаться завершения других, прежде чем начать или продолжить свою работу. Пример такой ситуации – проблемы вида “*map-reduce*”. Для решения подобных проблем сначала выполняются задачи “*map*”, обрабатывающие входные данные, а затем задачи “*reduce*”, которые агрегируют (или свертывают) полученные на предыдущем шаге данные в некое результирующее значение.

В большинстве случаев задачи обработки хорошо поддаются переносу в многозадачную среду, тогда как задачи свертки – гораздо хуже (хотя можно при помощи нескольких задач произвести промежуточную свертку, а затем при помощи одной задачи выполнить окончательную свертку для получения итогового результата). Очевидно, что задачи свертки должны выполняться только по завершении всех или части задач по обработке. Ана-

логично, если мы используем промежуточную свертку, то задача по окончательной свертке должна выполняться только по завершении задач промежуточной свертки данных. Вот пример проблемы, решаемой при помощи подхода “*map-reduce*”: пусть нас интересует частота использования слов в большом тексте. Тогда при помощи задач обработки мы могли бы посчитать частоту использования слов в каждом параграфе (и эти задачи будут хорошо работать параллельно), после чего при помощи задачи свертки получить итоговый результат.

Видно, что между разными задачами существует определенное упорядочение выполнения: одни задачи начинают свою работу только тогда, когда другие задачи закончат свою. Чтобы обеспечить это упорядочение, используются средства синхронизации между задачами.

Синхронизация между задачами нужна и тогда, когда у нас есть данные, доступ к которым имеют несколько задач одновременно (если к некоторым данным имеет доступ только одна задача, беспокоиться не о чём). Если все задачи только читают данные, никакой синхронизации между ними не требуется. Проблемы начинаются, когда какие-либо из задач начинают изменять данные.

Рассмотрим пример на языке C, иллюстрирующий данную проблему. Пусть у нас есть глобальная переменная **X**, доступ к которой имеют несколько задач, и следующий простой блок кода, изменяющий наши данные (глобальную переменную **X**): **{X *= 3;}**. Для понимания проблем, возникающих при выполнении этого блока кода, давайте опустимся на уровень ниже и рассмотрим один из его аналогов на языке ассемблера (для процессоров x86):

```
imul eax, [X], 3  
mov dword ptr [X], eax
```

По поводу этого кода на ассемблере следует сделать два замечания. Во-первых, это всего лишь один из аналогов приведенного выше блока кода; мы не можем гарантировать, какие инструкции в действительности генерирует компилятор. Во-вторых, обычно перед использованием того или иного регистра его сохраняют в стеке, а после использования восстанавливают его предыдущее значение (при помощи инструкций **push** и **pop**); мы сознательно пропускаем эти инструкции (но помним, что они есть).

Но вернемся к нашему примеру: предположим, что две задачи одновременно выполняют этот блок кода (например, на разных ядрах процессора), и рассмотрим один из возможных вариантов одновременного выполнения. Для определенности, пусть значение переменной **X** до выполнения этого блока кода было **3**. На первом шаге обе задачи начинают выполнение этого блока кода (входят в него). На следующем шаге обе задачи выполняют инструкцию **imul**: в результате для обеих задач значение регистра **eax** становится **9**. На следующем шаге обе задачи сохраняют свои

значения регистра **eax** в ячейку памяти, связанной с переменной **X**; в результате переменная **X** получает значение **9**. И на последнем шаге обе задачи заканчивают выполнение этого блока кода (выходят из него).

Если бы две эти задачи выполняли этот блок кода последовательно, переменная **X** получила бы значение **27**, а в нашем примере переменная **X** получила значение **9**.

В этом примере мы показали проблему, известную как состязание (гонка) за ресурсы. Другая известная проблема, связанная с одновременным изменением данных – повреждение данных. Обычно данные повреждаются, когда несколько задач одновременно обновляют сложные структуры данных, доступ к которым не атомарен на уровне процессора. Например, на платформе x86 такой структурой данных будут 64-битные целые числа. После такого обновления в подобной структуре могут содержаться данные всех процессов, производивших обновление, и, соответственно, сама структура содержит данные, которые в ней не появились бы, если бы все процессы обновляли ее последовательно. Возникает вполне логичный вопрос: что надо делать, чтобы подобных ситуаций не возникало? Ответ вполне очевиден: если несколько процессов одновременно обращаются к некоторым данным, причем некоторые из этих процессов изменяют общие данные, то для доступа к этим данным необходимо использовать средства синхронизации.

«Между задачами существует упорядочение выполнения.»

О синхронизации

Следует сказать, что на многоядерных и многопроцессорных машинах возможна гонка за ресурсы, связанная с тем, что разные ядра (или процессоры) в своей каш-памяти содержат разные значения одной и той же переменной. Это справедливо, даже если доступ к такой переменной атомарен на уровне процессора, как, например, для 32-битных целых чисел на платформе x86. Следует учитывать подобные ситуации и применять соответствующие средства синхронизации, чтобы избежать их.

Теперь рассмотрим, какие средства для синхронизации задач у нас есть. Не будем сейчас говорить о конкретных средствах; займемся средствами для синхронизации задач в общем.

Существуют два класса средств синхронизации: это средства, которые могут приводить к блокировкам задач (синхронизация с блокировками) и средства, которые к блокировкам задач не приводят (неблокирующая синхронизация). Работа средств синхронизации с блокировками основана на специальном объекте, называемом блокировкой. Объект, представляющий блокировку, обладает несколькими (минимум двумя) состояниями, и его поведение меняется в зависимости от того, в каком состоянии он находится. В простейшем случае такой объект имеет два состояния; например, для мьютекса это «свободен» и «занят».

Когда объект блокировки находится в свободном состоянии, любая задача может «захватить» его (при помощи функции из API); при этом объект блокировки перейдет в занятое состояние. Когда какая-то другая задача попытается «захватить» объект блокировки, находящийся в занятом состоянии, выполнение этой задачи будет заблокировано (и она перейдет в состояние ожидания) до тех пор, пока объект блокировки не перейдет в свободное состояние. Когда задача, владеющая объектом блокировки, «освободит» его (при помощи функции из API), любая другая задача, ожидающая освобождения этого объекта блокировки, может «захватить» его. Обычно при этом со всех задач снимается блокировка, после чего какая-то одна из задач «захватывает» объект блокировки, а все остальные задачи блокируются (при попытке «захватить» этот объект блокировки). В более сложных случаях и поведение объекта блокировки будет более сложным: объект блокировки может разрешать «захватывать» себя некоторым задачам (например, когда объект блокировки представляет собой семафор или блокировку чтения-записи), может применяться для сигнализации о некотором событии (условные переменные в POSIX, объекты ядра, событие в WIN32 API) и т. д.

Работа средств неблокирующей синхронизации основана на таких средствах, как атомарные операции и специальные механизмы блокировки. Эти специальные механизмы блокировки не блокируют задачу (не переводят ее в состояние ожидания), если объект блокировки не может быть «захвачен» данной задачей. Вместо этого задача в бесконечном цикле проверяет, не освободился ли этот объект блокировки (т. н. спин-блокировка). Атомарная операция – это операция, которая выполняется атомарно на процессоре, т. е. выполнение задачи может быть прервано либо до, либо после такой операции. К подобным операциям относятся инкремент, декремент, сравнение с обменом (CAS) и др. Наиболее значимая из атомарных операций – операция сравнения с обме-

Реентерабельность блокировок

Реентерабельность – это возможность повторного использования какого-либо объекта или вызова функции в момент, когда данный объект используется или функция вызвана. В случае объектов блокировки это означает, может ли одна и та же сторона «захватить» объект блокировки несколько раз. Если да, то такой объект блокировки явля-

ется реентерабельным (при этом, если мы N раз «захватили» объект блокировки, его необходимо «освободить» также N раз); если же нет – нереентерабельным (при попытках «захватить» такой объект блокировки несколько раз мы в итоге получим самоблокировку). Блокировки, поддерживаемые модулем `global`, являются реентерабельными.

ном, которая атомарно проверяет значение переменной с некоторым заданным значением и при несовпадении устанавливает значение переменной в заданное.

О необходимости применения синхронизации и, соответственно, о том, что нам дает синхронизация задач, мы уже поговорили. Давайте посмотрим теперь на то, какую цену мы за использование синхронизации платим (как известно, мы за все платим какую-то цену, т. к. бесплатный сыр бывает только в мышеловке).

Первая и наиболее очевидная плата за использование средств синхронизации – усложнение исходного кода приложения (по сравнению с вариантом без использования средств синхронизации или однозадачным вариантом). Более того, приложение, разработанное с использованием средств неблокирующей синхронизации, обычно имеет более сложную структуру по сравнению с аналогичным приложением, разработанным с использованием средств синхронизации с блокировками. Использование этих средств приводит к снижению производительности (временами значительному) в ситуации многозадачности на одном многоядерном или многопроцессорном компьютере (обычно это затрагивает многозадачность на основе потоков). Связано это с тем, что при блокировании задачи она переходит в состояние ожидания, в результате чего остаток процессорного времени передается другой задаче; при этом происходит переключение контекста задачи, являющееся достаточно ресурсоемкой задачей. Если переключение контекста происходит достаточно часто, может получиться так, что приложение больше времени тратит на переключение контекста, чем на выполнение своих задач.

Со средствами синхронизации с блокировками связана также такая большая проблема, как взаимные блокировки задач. В простейшем случае взаимные блокировки задач получаются следующим образом. Пусть у нас есть два ресурса – **A** и **B**, доступ к которым защищается при помощи средств синхронизации с блокировками. Предположим, что у нас есть две задачи, которые хотят безопасно работать как с ресурсом **A**, так и с ресурсом **B**, при этом первая задача сначала пытается «захватить» блокировку, связанную с ресурсом **A**, а потом с ресурсом **B**, а вторая задача – наоборот. При одновременном выполнении этих задач возможна такая ситуация, когда первая задача «захватила» блокировку, связанную с ресурсом **A**, и ожидает блокировку,

»

» Не хотите пропустить номер? Подпишитесь на www.linuxformat.ru/subscribe/!

Блокировки на перекрывающихся подмножествах узлов

Функции `global:set_lock/1,2,3` и `global:del_lock/1,2` позволяют «захватывать» и «освобождать» блокировки, заданные идентификатором на множестве узлов, определяемых пользователем. Идентификатор блокировки – это пара, состоящая из идентификатора ресурса и идентификатора стороны, запрашивающей блокировку (в качестве идентификаторов могут выступать любые объекты языка Erlang). Может встать вопрос: а что будет, если

попытаться захватить один и тот же ресурс двумя разными сторонами на двух разных, но перекрывающихся подмножествах узлов? Например, мы делаем вызов `global:set_lock({res_id, side_id1}, ['n1@stdstring', 'n2@stdstring'], 0)`, который возвращает `true`. Что в таком случае вернет вызов `global:set_lock({res_id, side_id2}, ['n2@stdstring', 'n3@stdstring'], 0)`? Если бы мы запрашивали блокировку `{res_id, side_id2}` только на узле `n3@stdstring`.

мы, очевидно, ее «захватили» бы (и вызов вернул бы `true`). Однако мы запросили блокировку на узлах `'n2@stdstring'` и `'n3@stdstring'`; а значит, блокировки только на узле `'n3@stdstring'` нам не достаточно. Блокировку `{res_id, side_id2}` на узле `'n2@stdstring'` «захватить» нельзя (она «захвачена» другой стороной). Соответственно, нельзя захватить эту блокировку и на узлах `'n2@stdstring'` и `'n3@stdstring'`. Тогда второй вызов `global:set_lock/3` вернет `false`.

связанную с ресурсом **B**, тогда как вторая задача «захватила» блокировку, связанную с ресурсом **B**, и ожидает блокировку, связанную с ресурсом **A**. Такое ожидание будет вечным. Как мы говорили выше, средства синхронизации – это средства взаимодействия между процессами, и если разные задачи «захватывают» одни и те же блокировки в разном порядке, это означает, что какие-то из задач нарушают протокол взаимодействия между задачами.

Что касается средств неблокирующей синхронизации, то помимо сильного усложнения разрабатываемых приложений, далеко не все может быть реализовано только с их помощью. Так, например, двусвязный список не имеет реализации с использованием средств неблокирующей синхронизации. При использовании средств неблокирующей синхронизации ситуации взаимной блокировки задач возникнуть не может. Но вполне могут возникнуть ситуации зацикливания задач и гонки за ресурсы.

Теперь поговорим о синхронизации применительно к языку Erlang. Мы уже отмечали (см. [LXF158](#)), что создатели языка Erlang приняли решение максимально облегчить такую непростую область программирования, как многозадачность. Как результат, в языке Erlang есть всего один тип многозадачности – процессы языка Erlang, и средством взаимодействия (синхронизации) между ними являются сообщения. Заметьте, что процессы языка Erlang – это не то же самое, что процессы операционной системы: это всего лишь способ представления задач в языке Erlang. Обычно в одном экземпляре среды выполнения Erlang (являющейся процессом ОС) выполняется несколько процессов Erlang. Средство взаимодействия между процессами языка Erlang – обмен сообщений, который по сути является инкапсуляцией взаимодействия через сокеты. Минусы такого подхода тоже вполне очевидны. Большой объем данных в памяти с использованием процессов языка Erlang нельзя обработать так же эффективно, как с помощью нескольких потоков в одном процессе. Взаимодействие процессов посредством сообщений (и вообще функциональная природа языка Erlang) приводят к избыточному копированию данных (объектов, являющихся сообщениями) при передаче сообщений. И, наконец, в языке Erlang нет средств неблокирующей синхронизации.

Используемая в языке Erlang модель многозадачности была введена не только для упрощения разработки многозадачных приложений, но и для решения ряда проблем, вызванных использованием потоков в качестве задач. Понятно, что при таком подходе у нас не будет таких проблем, как гонка за ресурсы, или проблем, связанных с неблокирующей синхронизацией. Остается, пожалуй, один вопрос: возможна ли взаимная блокировка задач в языке Erlang? Мы уже говорили, что взаимная блокировка задач проявляется тогда, когда нарушается протокол взаимодействия между задачами (порядок «захвата» блокировок). У нас блокировок нет, но мы попробуем реализовать взаимную блокировку

задач, используя тот же принцип: одна из задач будет нарушать установленный протокол взаимодействия.

У читателей может возникнуть закономерный вопрос: а зачем нам пытаться реализовать взаимную блокировку задач? Ответ очевиден: зная, как это получается, мы, наверное, будем избегать такой ситуации. Давайте приступим к реализации: пусть первая задача посыпает сообщение **a** второй задаче и ожидает его же в ответ, после чего посыпает сообщение **b** и ожидает его же в ответ. А вторая задача делает все наоборот: ожидает сообщение **b** и посыпает его же обратно, после чего ожидает сообщение **a** и посыпает его обратно. Вот пример, реализующий это поведение:

```
fun1() ->
    receive
        {init, Process2} -> io:format("init message ~n", [])
    end,
    io:format("process 1, send message a ~n", []),
    Process2 ! {self(), a},
    receive
        {Process2, a} -> io:format("a message on process 1 ~n", [])
    end,
    io:format("process 1, send message b ~n", []),
    Process2 ! {self(), b},
    receive
        {Process2, b} -> io:format("b message on process 1 ~n", [])
    end.
fun2() ->
    receive
        {Process1, b} -> io:format("b message on process 2 ~n", [])
    end,
    io:format("process 2, send message b ~n", []),
    Process1 ! {self(), b},
    receive
        {Process1, a} -> io:format("a message on process 2 ~n", [])
    end,
    io:format("process 2, send message a ~n", []),
    Process1 ! {self(), a}.
```

Естественно, что функции `fun1/0` и `fun2/0` экспортirуются из некоторого модуля, например, из модуля `interlock_ex`. Так как функции будут основными телами двух независимых процессов, то первый процесс должен как-то узнать об идентификаторе второго процесса: для этого первый процесс после его создания ожидает сообщение вида `{init, Pid2}`, где `Pid2` – идентификатор второго процесса. Теперь давайте запустим наш пример и посмотрим на результирующий вывод (из функций `fun1/0` и `fun2/0`). Для этого запускаем среду выполнения Erlang, после чего создаем оба процесса: `Pid1 = spawn(fun interlock_ex:fun1/0)` и `Pid2 = spawn(fun interlock_ex:fun2/0)` (естественно, что модуль `interlock_ex` должен уже быть откомпилирован). И, наконец, осталось только со-

» **Пропустили номер?** Узнайте на с. 104, как получить его прямо сейчас.

общить первому процессу о втором, послав ему сообщение `linit, Pid2: Pid1!{init, Pid2}.`

В результате в консоли среды выполнения Erlang мы получим сообщения “`init message`” и “`process 1, send message a`” (среди сообщений будет также результат вычисления выражения `Pid1!{init, Pid2}`). Видно, что вместо полного цикла обмена сообщениями все закончилось на стадии отправления сообщения `a` первым процессом второму, т.е. налицо взаимная блокировка между этими двумя процессами. А ее причиной является нарушение протокола взаимодействия между процессами вторым процессом.

Как уже не раз говорилось, введение в язык Erlang многозадачности на основе процессов Erlang (которые работают изолированно друг от друга, пускай и в пределах одной среды времени выполнения), да и сама функциональная природа языка избавляет нас от таких проблем многозадачности, как гонка за ресурсы и повреждение данных. Это справедливо, пока мы работаем с ресурсами и данными, внутренними относительно среды выполнения Erlang; например, объекты языка Erlang являются такими ресурсами. Но стоит начать работать с внешними относительно среды времени выполнения Erlang ресурсами (например, с файлами), все перечисленные выше проблемы возвращаются. Действительно, попробуйте в двух задачах открыть один и тот же файл на запись и записать туда одну и ту же порцию данных одновременно; с большой долей вероятности вы увидите, что данные будут перемешаны. Поэтому, как только мы начинаем работать с внешними ресурсами, перед нами встают вопросы о защите этих ресурсов от одновременного доступа со стороны нескольких задач (кроме случая, когда все задачи ничего не изменяют в данных из внешнего ресурса). Давайте подробнее поговорим о том, как решаются подобные проблемы.

Наиболее легкий, простой и очевидный подход (он же и наиболее близкий к Erlang-way) к решению данной проблемы – использование сервис-ориентированной архитектуры (SOA). Действительно, если у нас есть внешний ресурс (например, файл), то давайте осуществлять к нему доступ не напрямую, а через некоторый сервис, взаимодействуя с ним при помощи отсылки запросов и получения ответов. Вполне логично, что таким сервисом будет процесс языка Erlang, а запросами и ответами будут сообщения (т.е. любые объекты языка). Тогда доступ к внешнему ресурсу будет осуществлять только этот сервисный процесс, и никаких проблем с одновременным доступом к ресурсу не возникнет. Если же в разных частях программы необходимо обращаться к нескольким разным файлам, то вполне логично, что в таком случае мы можем создать несколько экземпляров сервисов: по одному на каждый файл, с которым необходимо работать.

Описанный подход, при всех его достоинствах, не лишен и недостатков. Допустим, мы создали сервисы для работы с внешними ресурсами (например, файлом), но сейчас нам требуется обмен данными между двумя внешними ресурсами. Если мы можем прочитать все необходимые данные с ресурса источника за один раз, то обычно никаких проблем нет: сервисы доступа к ресурсам обеспечивают атомарность чтения и записи, а необходимость в атомарности операции обмена данными (чтения-записи) бывает не так уж часто. Однако если нельзя прочитать все данные с ресурса источника, или есть необходимость в атомарности операции обмена, необходимо что-то делать во избежание проблем. Первое, что приходит на ум, это сделать аналог блокировок: специальную пару сообщений, переводящую сервис для доступа к внешнему ресурсу в монопольный режим работы и обратно. К счастью, делать этого для каждого сервиса не надо: модуль `global` предоставляет средства реализации такой функциональности.

Для работы с такими блокировками используется понятие «идентификатор блокировки» – это кортеж из двух элементов: идентификатора ресурса и идентификатора стороны, запрашивающей блокировку. В качестве таких идентификаторов могут выступать любые объекты языка Erlang (существует достаточно небольшой список атомов, которые не рекомендуется использовать в качестве идентификаторов ресурсов).

Вполне возможна ситуация, что два разных процесса пытаются «захватить» блокировку на какой-то ресурс, используя один и тот же идентификатор запрашивающей стороны. В этом случае (естественно, если блокировка свободна) они оба ее «захватят»; однако и освобождать эту блокировку необходимо им обоим. Область действия этих блокировок – все известные узлы; однако область действия блокировок можно изменить, задав список узлов, для которых данная блокировка будет действительна. Если процесс, владеющей блокировкой, завершится без ее освобождения или же узел, на котором выполняется такой процесс, завершит свою работу, то блокировка автоматически освободится (если, конечно, ею никто больше не владеет).

После этого небольшого обзора взглянем на наших героев. Для «захвата» блокировки у нас есть следующие три функции: `global:set_lock(Id)`, `global:set_lock(Id, Nodes)` и `global:set_lock(Id, Nodes, Retries)`. Функция `global:set_lock/3` пытается установить блокировку с идентификатором `Id`, область действия которой распространяется на узлы `Nodes`, с количеством попыток установить блокировку `Retries`. В качестве значения для числа попыток установить блокировку можно передать любое неотрицательное число или атом `infinity`. Функция `global:set_lock/3` будет пытаться

«захватить» блокировку не более `Retries` раз, впадая на некоторое время в сон в случае неудачной попытки (если значением `Retries` является атом `infinity`, то функция `global:set_lock/3` будет выполняться, пока не «захватит» блокировку).

Эта функция вернет атом `true`, если блокировка была «захвачена», и `false` – в противном случае. Функция `global:set_lock/2` эквивалентна функции `global:set_lock/3` со значением `Retries`, равным атому `infinity`. Функция `global:set_lock/1` эквивалентна функции `global:set_lock/2`, только блокировка определяется на всех узлах. Для освобождения блокировки служат следующие две функции: `global:del_lock(Id)` и `global:del_lock(Id, Nodes)`. Функция `global:del_lock/2` позволяет освободить блокировку, заданную идентификатором `Id`, на узлах `Nodes`, а функция `global:del_lock/1` делает то же самое на всех узлах.

Сегодня мы познакомились поближе с таким явлением, как синхронизация задач (и с ее реализацией в языке Erlang). Мы увидели, что ничего страшного в синхронизации нет: достаточно быть аккуратным и соблюдать принятые протоколы взаимодействия между задачами. А в следующем номере мы начнем практикум, посвященный созданию многозадачных приложений. [LXF](#)

Блокировки: Упрощенный сценарий

Обычно работа с блокировками ресурсов выглядит так: мы «захватываем» блокировку, выполняем некую функцию (или последовательность действий, сводимую в некую функцию), после чего «освобождаем» блокировку. Конечно, мы можем не смыть «захватить» блокировку: тогда дальнейших действий не предвидится. Чтобы упростить этот сценарий, в модуле `global` определены функции `global:trans/2,3,4`. Функция `global:trans(Id, Fun, Nodes, Retries)` пытается захватить блокировку с идентификатором `Id` на узлах `Nodes` `Retries` раз. Если «захват» осуществлен, выполняется функция `Fun`, блокировка «освобождается» и возвращается результат выполнения функции `Fun`. Если «захватить» блокировку не удалось, возвращается атом `aborted`.