Распределенные информационно-аналитические системы

Лекция № 7.

Алгоритмы взаимного исключения

Профессор кафедры КБ-2: д.т.н. Шатовкин Р.Р.

Учебные цели:

Изучить основы научных знаний по общим положениям алгоритмов взаимных исключений; централизованному алгоритму; алгоритмам на основе получения разрешений; алгоритмам на основе передачи маркера.

Учебные вопросы:

1. Общие положения.

2. Централизованный алгоритм.

3. Алгоритмы на основе получения разрешений.

4. Алгоритмы на основе передачи маркера.

Процессам распределенной системы часто приходиться координировать свои действия. Например, они могут соревноваться за возможность работы с разделяемыми ресурсами, требующими эксклюзивного доступа. Для организации такой работы, во-первых, нужно в коде каждого процесса выделять некоторую его часть, называемую ***критической секцией*** (*critical section*), в которой есть обращения к таким ресурсам. Во-вторых, необходимо реализовать тот или иной механизм ***взаимного исключения*** (*mutual exclusion*), выражающийся в удовлетворении следующего основного требования: во время выполнения критической секции одного процесса с обращениями к разделяемым ресурсам ни один другой процесс не должен выполняться в своей критической секции, работающей с любым из этих ресурсов.

В некоторых случаях совместно используемые ресурсы управляются специализированными процессами (серверами), которые в том числе обеспечивают взаимное исключение при доступе к управляемым ресурсам со стороны других процессов (клиентов). Однако бывают ситуации, когда равноправные процессы должны координировать обращения к общему ресурсу самостоятельно между собой. В таких случаях требуются отдельные механизмы взаимного исключения, независимые от конкретной схемы управления ресурсами.

Отсутствие общей памяти в распределенных системах не позволяет использовать разделяемые переменные (такие как семафоры) для решения рассматриваемой задачи: распределенные алгоритмы взаимного исключения должны опираться исключительно на обмен сообщениями между процессами. Разработка таких алгоритмов осложняется тем, что приходится иметь дело с произвольными задержками передачи сообщений и отсутствием полной информации о состоянии всей системы. Также не делается никаких предположений об относительной скорости выполнения процессов и об их количестве.

В данной лекции мы рассмотрим основные распределенные алгоритмы взаимного исключения, ключевые идеи которых используются и для решения многих других задач в распределенных системах. Кроме того, изучение этих алгоритмов позволяет раскрыть такие важные вопросы, как обеспечение свойств безопасности и живучести распределенных алгоритмов.

1. Общие положения

При рассмотрении распределенных алгоритмов взаимного исключения мы будем оставаться в рамках следующих допущений: процессы выполняются и взаимодействуют асинхронно, ни один из процессов не выходит из строя, все каналы связи являются надежными, и коммуникационная сеть не разбивается на несвязанные друг с другом части.

Мы не будем делать никаких предположений о том, являются ли каналы между процессами очередями (FIFO) или нет (non-FIFO). Требование к свойству очередности каналов будет определяться отдельно для каждого обсуждаемого алгоритма.

Также мы будем полагать, что обращения к разделяемым ресурсам происходят из одной единственной критической секции. Распространение представленных ниже алгоритмов на случаи работы с более чем одной критической секцией не вызывает затруднений.

С позиции организации работы с разделяемыми ресурсами в коде каждого процесса можно выделить следующие части:

(1) критическая секция, в которой есть обращения к разделяемым ресурсам;

(2) предшествующая и последующая часть кода, где такие обращения отсутствуют. Общая структура кода процессов распределенной системы, рассматриваемая далее для изучения механизмов взаимного исключения, приведена в Листинге 1.

Листинг 1. Структура кода с критической секцией.

P_1		P_N
`void P1(int R)`		`void PN(int R)`
`{`	…	`{`
`while(true)`		`while(true)`
`{`		`{`
`/* предшествующий код */;`		`/* предшествующий код */;`
`request_cs(R);`		**`request_cs(R);`**
`/* критическая секция */;`		`/* критическая секция */;`
`release_cs(R);`		**`release_cs(R);`**
`/* последующий код */;`		`/* последующий код */;`
`}`		`}`
`}`		`}`

В коде каждого процесса используются две функции: request_CS() для получения разрешения на вход в критическую секцию и блокировки процесса в случае необходимости и release_CS() для выхода из критической секции и предоставления возможности остальным процессам войти в нее. Каждая из этих функций принимает в качестве аргумента общий для всех процессов идентификатор критической секции R. Суть разработки распределенных алгоритмов взаимного исключения как раз и состоит в построении механизмов, обеспечивающих работу этих двух функций.

Исходя из организации работы с критической секцией множество возможных состояний процесса будет определяться тремя состояниями:

(1) запрос на вход в критическую секцию;

(2) выполнение внутри критической секции;

(3) выполнение вне критической секции.

При этом возможны только переходы:

- из состояния выполнения вне критической секции в состояние запроса на вход в критическую секцию (вызов функции request_CS);

- из состояния запроса на вход в критическую секцию в состояние выполнения внутри критической секции (возврат из функции request_CS);

- из состояния выполнения внутри критической секции вновь в состояние выполнения вне критической секции (вызов функции release_CS и возврат из нее).

Важно отметить, что в состоянии выполнения внутри критической секции процесс может находиться лишь ограниченное время, в то время как в состоянии выполнения вне критической секции процесс может находиться сколь угодно долго. В состоянии запроса на вход в критическую секцию процесс блокируется до получения разрешения на вход в критическую секцию и не должен формировать новых запросов для доступа к критической секции.

Основные требования к алгоритмам взаимного исключения формулируются следующим образом.

Безопасность. В каждый момент времени в критическую секцию может выполняться не более одного процесса.

Живучесть. Каждый запрос на вход в критическую секцию рано или поздно должен быть удовлетворен.

Условие живучести алгоритма взаимного исключения говорит о том, что нельзя допускать ситуации ***взаимной блокировки***, то есть бесконечного ожидания прихода сообщения, которое никогда не придет, и ситуации ***голодания***, то есть бесконечного ожидания разрешения на вход в критическую секцию одним процессом, в то время как другие процессы неоднократно получают доступ к критической секции.

Отсутствие голодания также является условием ***справедливости***. Другой контекст определения справедливости обслуживания запросов на вход в критическую секцию связан с установлением порядка вхождения процессов в критическую секцию и ограничением на число позволяемых входов-выходов в критическую секцию для одного процесса, пока другой процесс ожидает входа в критическую секцию. А именно, логично потребовать, что в случае, если один запрос на вход в критическую секцию произошел раньше другого, то и доступ к критической секции должен быть предоставлен в таком же порядке. Если механизм взаимного исключения удовлетворяет этому требованию, и все запросы на вход в критическую секцию связаны отношением причинно-следственного порядка, то ни один процесс не сможет войти в критическую секцию более одного раза, пока другой процесс ожидает получения разрешения на вход в критическую секцию.

Все многообразие алгоритмов взаимного исключения в распределенных системах на самом деле базируется на реализации одного из ***двух основных принципов***:

1. ***Алгоритмы на основе получения разрешений (permission-based algorithms).*** В этом случае для входа в критическую секцию процессу требуется собрать «достаточное» число разрешений от других процессов распределенной системы. Свойство безопасности будет выполнено, если такое число разрешений сможет получить лишь один процесс из всех процессов, желающих войти в критическую секцию. Свойство живучести обычно обеспечивается за счет упорядочивания запросов по их отметкам логического времени или с помощью ациклического графа предшествования, в котором вершины соответствуют процессам распределенной системы, а направленные ребра описывают приоритет процессов друг над другом для определения порядка предоставления доступа к критической секции.

2. ***Алгоритмы на основе передачи маркера (token-based algorithms).*** Для таких алгоритмов право войти в критическую секцию 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 в этом случае будет гарантировано ввиду уникальности маркера. Поэтому основные усилия при разработке алгоритмов данного класса направлены на обеспечение свойства живучести, а именно, на управление перемещением маркера таким образом, чтобы он рано или поздно оказался в каждом процессе, желающем войти в критическую секцию.

Существуют ***два подхода к решению этой задачи***:

- ***непрерывное перемещение маркера среди всех процессов распределенной системы (perpetuum mobile)***;

- ***перемещение маркера лишь в ответ на получение запроса от заинтересованного в нем процесса (token asking methods)***.

Отметим, что при формировании запросов на владение маркером необходимо создать такие условия их распространения, при которых каждый запрос рано или поздно достигнет процесса, в котором находится маркер, вне зависимости от перемещений самого маркера.

Обратим внимание, что два перечисленных принципа построения алгоритмов взаимного исключения объединяются в централизованном алгоритме, когда доступ к критической секции управляется специально выделенным процессом, называемым координатором. В этом случае для входа в критическую секцию процессу достаточно получить одно единственное разрешение от координатора, который становится ответственным за поддержку информации о состояниях всех процессов и за выдачу разрешений на вход в критическую секцию таким образом, чтобы гарантировать выполнение свойств безопасности и живучести. С другой стороны, такое уникальное разрешение, выдаваемое координатором, можно рассматривать как маркер, передаваемый координатором процессу, желающему войти в критическую секцию, и возвращаемый обратно координатору после его использования. Более подробно работу централизованного алгоритма взаимного исключения мы рассмотрим в следующем подразделе.

Централизованные и распределенные алгоритмы. Централизованный алгоритм предполагает, что основная часть работы выполняется одним процессом или небольшой группой процессов (в сравнении с их общим числом), в то время как остальные процессы играют незначительную роль в достижении общего результата. Обычно эта роль сводится к получению информации от главного процесса или предоставлению ему нужных данных. Поэтому централизованный алгоритм всегда является ***асимметричным***: различные процессы выполняют логически разные функции, хотя, возможно, в чем-то и совпадающие.

Наиболее подходящей для централизованных алгоритмов архитектурой вычислительных систем является архитектура «клиент-сервер», в которой сервер владеет и распоряжается информационными ресурсами, а клиент имеет возможность воспользоваться ими. Большинство коммерческого программного обеспечения разрабатывается именно по такой архитектуре.

Отметим, что с теоретической точки зрения централизованный сервер потенциально является узким местом всей системы как из-за ограничений в собственной производительности, так и из-за ограничений в пропускной способности его линий связи. Кроме того, такой сервер становится единой точкой отказа распределенной системы.

На практике перечисленные проблемы чаще всего решаются путем использования реплицированных серверов, разнесенных друг от друга территориально. Однако такую конфигурацию уже нельзя считать полностью централизованной.

В распределенных алгоритмах каждый процесс играет одинаковую роль в достижении общего результата и в разделении общей нагрузки. Поэтому распределенные алгоритмы являются ***симметричными***: все процессы исполняют один и тот же код и выполняют одни и те же логические функции.

На самом деле абсолютно симметричные алгоритмы представляются практически идеальной конструкцией и для реализации в реальных системах более предпочтительными являются частично распределенные алгоритмы, обладающие естественной асимметрией в функциях своих узлов. К примеру, если процессы распределенной системы логически организованы в дерево, то его корень и листья обычно выполняют немного разные функции, при этом отличающиеся от функций вершин ветвления. Однако отметим, что возрастающая популярность мобильных систем, одноранговых (*peer-to-peer*) и самоорганизующихся (*ad-hoc*) сетей будет требовать полностью распределенных решений.

2. Централизованный алгоритм

Наиболее простой способ организации взаимного исключения в распределенных системах состоит в том, что один процесс выбирается координатором, который единолично предоставляет разрешение на вход в критическую секцию. Таким процессом, например, может быть процесс с наибольшим идентификатором.

Каждый раз, когда процесс распределенной системы собирается войти в критическую секцию, он посылает координатору сообщение REQUEST (ЗАПРОС) с соответствующим запросом и ожидает получения разрешения.

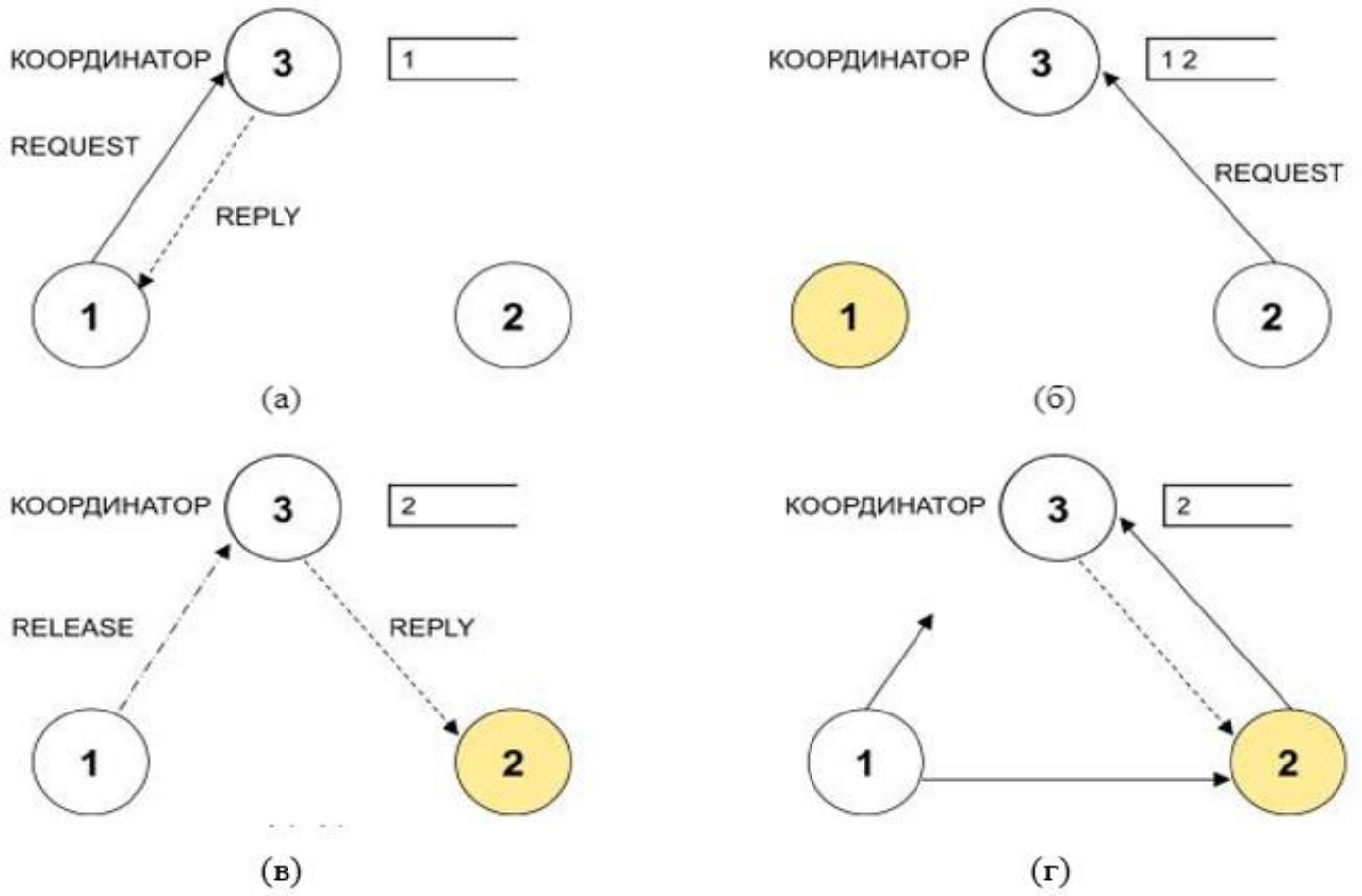


Рисунок 1 – Пример работы централизованного алгоритма взаимного исключения

Поступающие координатору запросы помещаются в очередь согласно порядку их поступления. Если на момент получения запроса ни один из процессов не находится в критическую секцию, то есть поступивший запрос занимает первое место в очереди, координатор немедленно посылает ответ REPLY (ОТВЕТ) с разрешением на вход в критическую секцию. После получения ответа процесс, запросивший доступ, входит в критическую секцию.

Эта ситуация проиллюстрирована на рисунке 1а: процесс P1 запрашивает вход в критическую секцию у координатора P3 и получает ответ; очередь координатора P3 изображена прямоугольником.

Предположим теперь, что пока процесс P1 выполняется в критическую секцию, другой процесс P2 запрашивает у координатора разрешение на вход в критическую секцию (см. рисунок 1б; выполняющийся в критической секции процесс P1 отмечен серым цветом).

Так как очередь не пуста, координатор знает, что в критической секции уже находится процесс P1, и не дает процессу P2 разрешения на вход. Конкретный способ запрещения доступа зависит от особенностей системы.

На рисунке 1б координатор просто не отвечает, тем самым блокируя процесс P2 в состоянии запроса на вход в критическую секцию. Также, координатор может послать ответ, гласящий «доступ запрещен». В любом случае запрос от P2 помещается в конец очереди координатора.

Когда процесс P1 выходит из критической секции, он отправляет координатору сообщение RELEASE (ОСВОБОЖДАЮ), тем самым предоставляя остальным процессам возможность войти в критическую секцию.

При получении сообщения RELEASE координатор удаляет запрос процесса P1 из своей очереди и отправляет разрешающее сообщение процессу, запрос которого окажется первым в его очереди; в нашем примере – процессу P2. Увидев разрешение, P2 войдет в критическую секцию, как показано на рисунке 1в.

Легко заметить, что представленный алгоритм гарантирует свойство безопасности: координатор позволяет войти в критическую секцию только одному процессу за раз. Кроме того, никакой процесс никогда не ждет разрешения на вход в критическую секцию вечно, так как запросы обслуживаются координатором согласно порядку их поступления. Поэтому алгоритм также удовлетворяет условию живучести.

Однако, строго говоря, такая схема не обеспечивает выполнение одного из аспектов свойства справедливости, согласно которому запросы на вход в критическую секцию должны удовлетворяться в порядке их возникновения в системе, а не в порядке их получения координатором. Как следствие, становится возможной ситуация, в которой один процесс сможет несколько раз войти и выйти из критической секции, в то время как другой процесс ожидает входа в критическую секцию, даже при условии того, что запрос от второго процесса произошел раньше всех запросов от первого процесса.

Рассмотрим, например, случай, когда процесс P1 отправляет запрос на вход в критическую секцию координатору P3 и после этого отправляет сообщение процессу P2. После получения сообщения от P1 процесс P2 отправляет координатору P3 свой запрос на вход в критическую секцию. Очевидно, что если запрос от P2 достигнет P3 раньше запроса от P1, то и доступ к критической секции будет предоставлен P2 вперед P1 (см. рисунок 1г).

Этот пример иллюстрирует тот факт, что порядок поступления запросов к координатору может отличаться от причинно-следственного порядка их возникновения в системе.

Тем не менее, описанный алгоритм прост в реализации и использует для работы с критической секцией всего 3 сообщения: для входа в критическую секцию требуется два сообщения – REQUEST и REPLY, для выхода из критической секции – одно сообщение RELEASE.

Недостаток алгоритма: обычный недостаток централизованного алгоритма – низкая отказоустойчивость (возможный отказ координатора или его перегрузка сообщениями).

3. Алгоритмы на основе получения разрешений

Распределенный алгоритм Лэмпорта. Алгоритм, предложенный Л. Лэмпортом, является одним из самых ранних распределенных алгоритмов взаимного исключения и призван проиллюстрировать применение скалярных часов для линейного упорядочивания операций, производимых процессами в распределенной системе, таких, например, как вход в критическую секцию и выход из критической секции.

В его основе лежат предположения о том, что все каналы связи обладают свойством FIFO, и сеть является полносвязной. Данный алгоритм обобщает рассмотренный выше централизованный алгоритм, в котором запросы на вход в критическую секцию помещаются в очередь и обслуживаются из нее по порядку, в том смысле, что позволяет рассматривать такую очередь в виде отдельного объекта, разделяемого между всеми процессами, а не находящегося под управлением одного единственного процесса (координатора).

Важным преимуществом алгоритма Лэмпорта является также то, что запросы на вход в критическую секцию обслуживаются не в произвольном порядке, как в централизованном алгоритме, а в порядке их возникновения в системе, то есть в порядке, определяемом их отметками логического времени. Поэтому, если один запрос на вход в критическую секцию произошел раньше другого, то и доступ к критической секции по первому запросу будет предоставлен раньше, чем по второму.

Здесь под отметками времени запросов на вход в критическую секцию мы будем подразумевать упорядоченные пары (Li, i), где Li – скалярное время процесса Pi на момент формирования запроса, i – идентификатор процесса Pi.

Алгоритм Лэмпорта опирается на принцип: чтобы реализовать представление общей совместно используемой очереди, каждый процесс работает с ее локальной «копией», и для определения единого для всех процессов порядка обслуживания запросов использует их отметки логического времени, упорядоченные линейно.

Чтобы каждый запрос оказался в каждой из таких локальных «копий», всякий раз, когда процесс собирается войти в критическую секцию, он помещает свой запрос вместе с отметкой времени в свою локальную очередь и рассылает сообщение с этим же запросом всем остальным процессам.

При получении запроса остальные процессы помещают его уже в свои локальные очереди. По порядку обслуживания запросов процесс получит право войти в критическую секцию, когда значение отметки времени его запроса окажется наименьшим среди всех других запросов. Остается лишь ответить на вопрос: как процессу убедиться в том, что его запрос имеет наименьшую отметку времени?

Действительно, из-за задержек передачи сообщений, возможна ситуация, когда в локальных очередях двух различных процессов в течение некоторого временного интервала наименьшей отметкой времени будут обладать разные запросы, что может привести к нарушению требования взаимного исключения.

Поэтому прежде чем процесс примет решение о входе в критическую секцию, исходя из состояния своей собственной очереди, он должен получить от всех других процессов сообщения, гарантирующие, что в каналах не осталось ни одного сообщения с запросом, имеющим время меньшее, чем его собственный запрос.

Благодаря свойству FIFO каналов связи и правилам работы логических часов такими сообщениями могут быть ответные сообщения, высылаемые при получении сообщения с запросом, или же любые другие сообщения, направляемые процессу, запрашивающему вход в критическую секцию, с отметкой времени большей, чем отметка времени его запроса.

Обозначим через Qi локальную очередь процесса Pi, в которую помещаются все запросы на доступ к критической секции вместе с их отметками времени. Тогда алгоритм Лэмпорта будет определяться следующими правилами.

Запрос на вход в критическую секцию:

Когда процессу Pi нужен доступ к критической секции, он рассылает сообщение REQUEST(Li, i) со значением своего логического времени (Li, i) всем остальным процессам и, кроме того, помещает этот запрос в свою очередь Qi. Каждая такая рассылка представляет собой одно атомарное событие процесса Pi. Когда процесс Pj получает запрос REQUEST(Li, i) от процесса Pi, он помещает его в свою очередь Qj и отправляет процессу Pi ответ REPLY(Lj, j) со значением своего логического времени (Lj, j).

Вход в критическую секцию:

Процесс Pi может войти в критическую секцию, если одновременно выполняются два условия.

1. Запрос REQUEST(Li, i) процесса Pi обладает наименьшим значением отметки времени среди всех запросов, находящихся в его собственной очереди Qi.

2. Процесс Pi получил сообщения от всех остальных процессов в системе с отметкой времени большей, чем (Li, i). При условии того, что каналы связи обладают свойством FIFO, соблюдение этого правила гарантирует, что процессу Pi известно обо всех запросах, предшествующих его текущему запросу.

Выход из критической секции:

При выходе из критической секции процесс Pi удаляет свой запрос из собственной очереди Qi и рассылает всем другим процессам сообщение RELEASE(Li, i) с отметкой своего логического времени.

По-прежнему каждая такая рассылка представляет собой одно атомарное событие процесса Pi. Получив сообщение RELEASE(Li, i), процесс Pj удаляет запрос процесса Pi из своей очереди Qj. После удаления процессом Pj запроса процесса Pi из Qj отметка времени его собственного запроса может оказаться наименьшей в Qj, и Pj сможет рассчитывать на вход в критическую секцию.

Обратим внимание, что представленный алгоритм является распределенным: все процессы следуют одним и тем же правилам, и каждый процесс принимает решение о входе в критическую секцию только на основе своей локальной информации.

Также отметим, что когда получение всех сформированных запросов на доступ к критической секции подтверждено всеми процессами, все локальные очереди будут пребывать в одинаковом состоянии, что как раз и позволяет рассматривать их как «копии» некоторой отдельной очереди, разделяемой между процессами. Поэтому можно сказать, что решение на вход в критическую секцию принимается на основе локальной информации, которая, однако, глобально согласована.

Легко увидеть, что алгоритм Лэмпорта обладает свойствами безопасности и живучести.

Докажем выполнение свойства безопасности от противного. Пусть два процесса Pi и Pj выполняются в критической секции одновременно. Это означает, что отметка времени запроса Pi оказалась наименьшей в его очереди Qi, а отметка времени запроса Pj – наименьшей в очереди Qj, и при этом оба процесса получили друг от друга ответные сообщения со временем большим, чем время их запросов. Без потери общности будем считать, что значение времени запроса Pi меньше времени запроса Pj. Тогда, опираясь на свойство FIFO каналов связи и правила работы логических часов, можно утверждать, что в момент получения процессом Pj ответного сообщения от Pi в очереди Qj уже должен был находиться запрос процесса Pi, и запрос Pj не мог иметь наименьшую отметку времени в Qj.

Покажем теперь, что алгоритм Лэмпорта обладает свойством живучести. Действительно, механизм ответных сообщений гарантирует, что любой процесс Pi, запрашивающий доступ к критической секции, рано или поздно получит от всех других процессов сообщения с отметкой времени большей, чем время его запроса. Поэтому второе условие входа в критическую секцию рано или поздно будет выполнено.

Кроме того, перед запросом Pi в очереди может оказаться не более (N – 1) запросов от других процессов с меньшими значениями отметки времени. При выходе из критической секции процесс Pj с наименьшим временем запроса разошлет сообщение «RELEASE», которое приведет к удалению запроса Pj из очередей всех процессов, в том числе из очереди Qi. Последующие запросы на вход в критическую секцию от Pj будут иметь время большее, чем время запроса Pi, и будут обслужены после Pi. Поэтому за ограниченное число шагов отметка времени запроса Pi окажется наименьшей в Qi и первое условие входа в критическую секцию также окажется выполненным. Следовательно, любой процесс, запрашивающий доступ к критической секции, в конце концов, сможет войти в нее.

Также отметим, что доступ к критической секции предоставляется строго в соответствии с отметками времени запросов по порядку, сохраняющему частичный причинно-следственный порядок →. Поэтому можно утверждать, что запросы на вход в критическую секцию обслуживаются в порядке их возникновения в системе.

Пример работы распределенного алгоритма Лэмпорта приведен на рисунке 2.

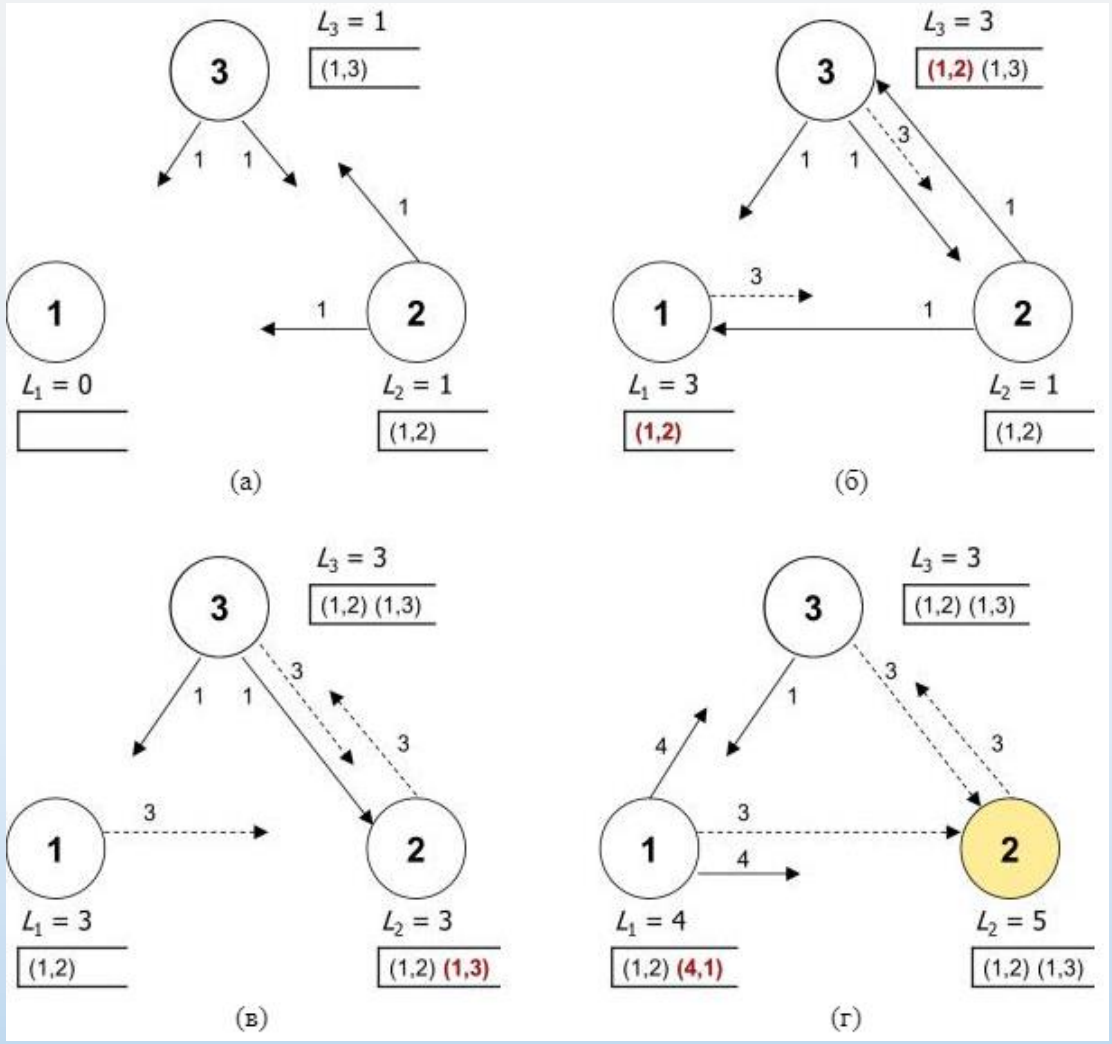
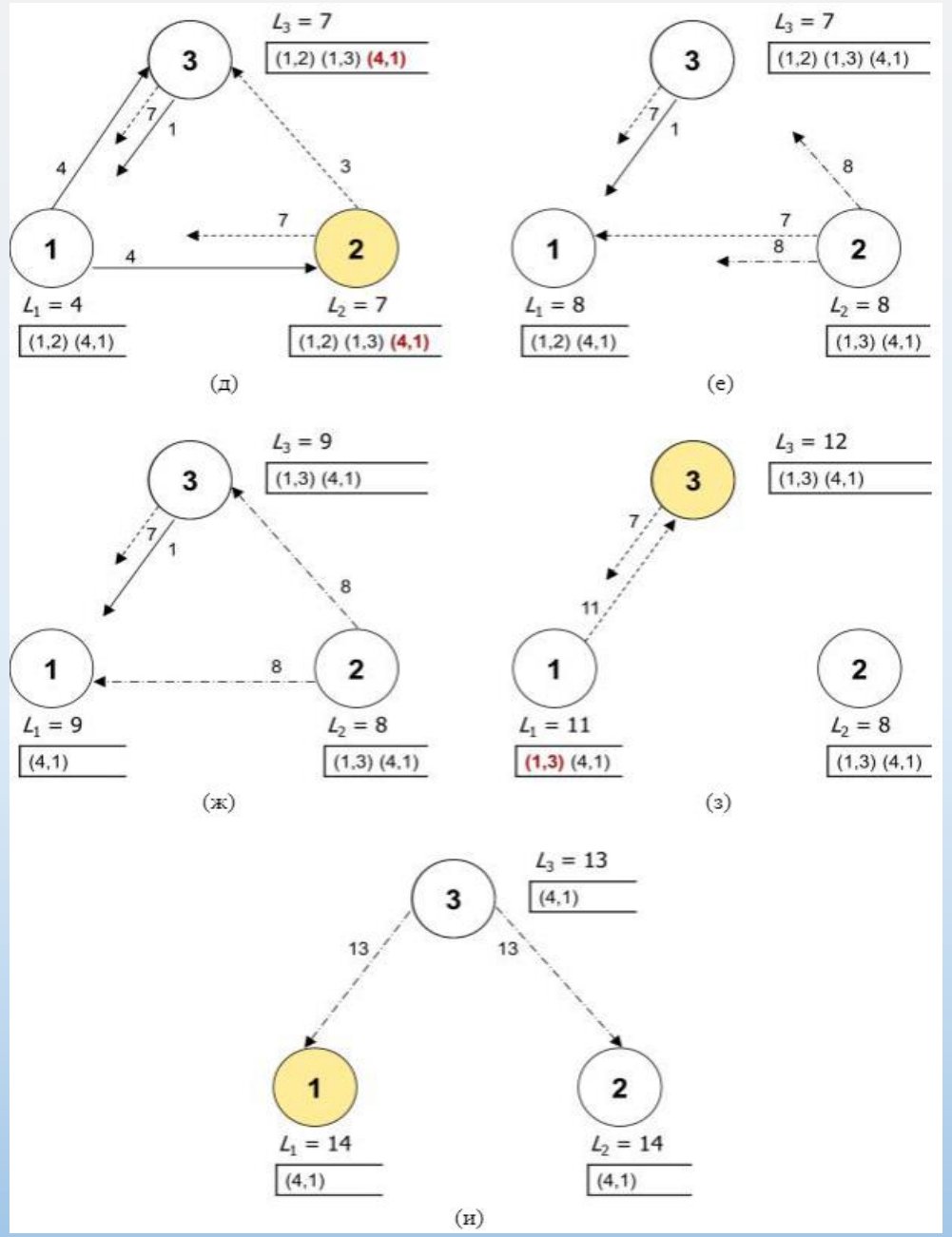


Рисунок 2 – Пример работы распределенного алгоритма Лэмпорта



На рисунке 2а приблизительно в одно и то же время процессы P2 и P3 запрашивают вход в критическую секцию, изменяя показания своих скалярных часов L2 и L3; локальная очередь каждого процесса изображена прямоугольником (будем считать, что в очередях запросы сортируются в порядке возрастания их отметок времени).

На рисунке 2б процессы P1 и P3 получают запрос от P2 и отправляют ему ответные сообщения. Так как отметка времени запроса P2 меньше отметки времени запроса P3, запрос от P2 помещается перед запросом от P3 в его собственной очереди Q3. Это дает возможность P2 войти в критическую секцию вперед P3. При этом процесс P2 сможет рассчитывать на вход в критическую секцию только тогда, когда получит ответные сообщения от P1 и P3.

Отметим, что благодаря свойству FIFO каналов связи, P2 не сможет получить ответное сообщение от P3 вперед запроса, отправленного ранее процессом P3. Поэтому вначале P2 получит запрос от P3, поместит его в свою локальную очередь Q2 и отправит P3 ответное сообщение.

Поскольку отметка времени поступившего от P3 запроса больше отметки времени запроса P2, запрос от P3 помещается в конец очереди Q2, как показано на рисунке 2в. Получив ответные сообщения от P1 и P3, процесс P2 войдет в критическую секцию, так как его запрос находится во главе его собственной очереди Q2 (рисунок 2г).

Далее в нашем сценарии процесс P1 переходит в состояние запроса на вход в критическую секцию и рассылает соответствующее сообщение остальным процессам. При получении запроса от P1 процессы P2 и P3 отправляют ответные сообщения (рисунок 2д).

Обратим внимание, что к этому моменту запрос на вход в критическую секцию от процесса P3 был получен и подтвержден процессом P2. Однако этот же запрос, направляемый процессу P1, так и не был получен.

На рисунке 2е процесс P2 выходит из критической секции, удаляет свой запрос из очереди Q2 и рассылает всем процессам сообщение RELEASE. Желающий войти в критическую секцию процесс P1 получает от P2 ответное сообщение и, затем, сообщение RELASE, что приводит к удалению запроса от P2 из очереди Q1 (рисунок 2ж).

Отметим, что на рисунке 2ж в состоянии запроса на вход в критическую секцию находится и процесс P1, и процесс P3. В локальной очереди каждого из этих процессов на первом месте расположен собственный запрос этого процесса. Оба этих процесса получили ответные сообщения от P2. Тем не менее, получить доступ к критической секции процесс P3 сможет раньше, чем P1, так как его запрос произошел раньше запроса P1.

Действительно, P1 не сможет войти в критическую секцию, пока не получит ответное сообщение от P3. Благодаря свойству FIFO канала связи между P3 и P1 это сообщение придет в P1 только после поступления запроса от P3, что проиллюстрировано на рисунке 2ж.

Когда же P1 получит запрос от P3, он поместит его вперед своего собственного запроса в своей очереди Q1, так как запрос от P3 имеет меньшую отметку времени, чем запрос от P1 (рисунок 2з). Теперь P1 не сможет получить доступ к критической секции, пока P3 не войдет и не выйдет из критической секции.

В свою очередь, P3 войдет в критическую секцию при получении ответного сообщения от P1 (рисунок 2з). При выходе из критической секции P3 разошлет всем процессам сообщение RELEASE, которое приведет к удалению его запроса из всех очередей, в том числе из очереди процесса P1. После этого запрос процесса P1 окажется первым в его собственной очереди Q1, и он сможет войти в критическую секцию, как показано на рисунке 2и.

Чтобы оценить эффективность работы алгоритма Лэмпорта, отметим, что для обеспечения взаимного исключения необходимо переслать 3(N – 1) сообщений: для входа в критическую секцию требуется (N – 1) сообщений REQUEST и (N – 1) ответных сообщений REPLY, для выхода из критической секции требуется (N – 1) сообщений RELEASE.

Недостаток: отказ любого процесса (парадоксально, но распределенный алгоритм оказывается менее отказоустойчивым, чем централизованный) приводит к отказу алгоритма, в целом.

Децентрализованный алгоритм на основе временных меток (алгоритм Рикарта-Агравала). Рассмотренный выше алгоритм Лэмпорта явным образом выстраивает запросы на доступ к критической секции в очередь в порядке возрастания их отметок времени.

Еще раз обратим внимание на роль сообщений REPLY и RELEASE в этом алгоритме. Ответные сообщения REPLY используются исключительно для информирования процесса, запрашивающего вход в критическую секцию, о том, что в каналах не осталось ни одного другого запроса, который мог бы встать перед его запросом в его локальной очереди. Роль сообщения RELEASE сводится к удалению обслуженного запроса из всех локальных очередей для предоставления другим процессам возможности войти в критическую секцию. Отсюда видно, что отметки логического времени, переносимые этими сообщениями, не играют существенной роли в функционировании алгоритма и могут быть опущены.

В этом случае при проверке второго условия входа в критическую секцию процесс будет опираться не на время получаемых сообщений, а на число поступивших сообщений REPLY: второе условие входа в критическую секцию в алгоритме Лэмпорта будет выполнено, когда процесс, запрашивающий доступ к критической секции, получит все сообщения REPLY от всех остальных процессов.

Г. Рикарт и А. Агравал предприняли попытку оптимизировать алгоритм Лэмпорта, исключив из него сообщения RELEASE за счет объединения функций сообщений RELEASE и REPLY в одном сообщений. Более того, алгоритм Рикарта-Агравала не требует, чтобы каналы связи обладали свойством FIFO.

Основная идея этого алгоритма заключается в том, что для входа в критическую секцию процессу требуется собрать разрешения от всех других процессов распределенной системы. Для этого процесс, желающий получить доступ к критической секции, рассылает свой запрос REQUEST всем остальным процессам и ожидает от них поступления разрешений в виде сообщений REPLY.

Условия, при которых процессы дают свое разрешение на вход в критическую секцию, сформулированы таким образом, чтобы удовлетворить требованиям безопасности и живучести. А именно, если все остальные процессы находятся в состоянии выполнения вне критической секции, они немедленно отправляют процессу, запрашивающему доступ к критической секции, сообщения REPLY.

Если же один из процессов выполняется в критической секции, то он откладывает отправку своего разрешения до тех пор, пока сам не покинет критическую секцию, тем самым, не позволяя двум процессам выполняться в критической секции одновременно.

Если же несколько процессов запрашивают разрешение на вход в критическую секцию в одно и то же время, то, как и в алгоритме Лэмпорта, для разрешения конфликта доступа к критической секции используются линейно упорядоченные отметки времени запросов (Li, i), где Li – скалярное время процесса Pi на момент формирования запроса, i – идентификатор процесса Pi.

Процесс, запрос которого имеет наименьшую отметку времени, получит право войти в критическую секцию первым. Это достигается за счет того, что каждый находящийся в состоянии запроса на вход в критическую секцию процесс Pi при получении запроса от другого процесса Pj сравнивает отметку времени (Li, i) своего запроса с отметкой времени (Lj, j) поступившего запроса, и откладывает отправку «REPLY», если (Li, i) < (Lj, j). Если же (Li, i) > (Lj, j), то процесс Pi отправляет сообщение «REPLY» процессу Pj немедленно. Поэтому процесс с наименьшей отметкой времени запроса получит ответные сообщения от всех остальных процессов и сможет войти в критическую секцию.

Для реализации алгоритма Рикарта-Агравала каждый процесс Pi поддерживает работу с массивом DRi[1..N], содержащим признаки отложенного ответа (от англ. deferred reply). Если процесс Pi откладывает отправку ответного сообщения процессу Pj, он устанавливает DRi[j] = 1; после отправки сообщения «REPLY» элемент DRi[j] сбрасывается в ноль. Все элементы DRi[k] инициализируется нулем, $1 \leq k \leq N$. С учетом этого массива алгоритм Рикарта-Агравала будет определяться следующими правилами:

Запрос на вход в критическую секцию:

Когда процессу Pi нужен доступ к критической секции, он переходит в состояние запроса на вход в критическую секцию, фиксируя показания своих скалярных часов Li, и рассылает сообщение REQUEST(Li, i) всем остальным процессам. Когда процесс Pj получает запрос REQUEST(Li, i) от процесса Pi, он отправляет процессу Pi ответ REPLY в случае, если Pj находится в состоянии выполнения вне критической секции, или в случае, если Pj находится в состоянии запроса на вход в критическую секцию и отметка времени (Lj, j) его запроса больше отметки времени (Li, i) запроса процесса Pi. В противном случае Pj откладывает отправку ответного сообщения REPLY и устанавливает DRj[i] = 1.

Вход в критическую секцию:

Процесс Pi может войти в критическую секцию, если он получил ответные сообщения REPLY от всех остальных процессов.

Выход из критической секции:

При выходе из критической секции процесс Pi рассылает отложенные сообщения «REPLY» всем ожидающим процессам, то есть всем процессам Pj, для которых DRi[j] = 1, и затем устанавливает DRi[j] = 0.

Представленный алгоритм удовлетворяет требованиям безопасности и живучести для решения задачи взаимного исключения.

Докажем ***выполнение свойства безопасности*** от противного. Пусть два процесса Pi и Pj выполняются в критической секции одновременно, и отметка времени запроса Pi меньше отметки времени запроса Pj.

Процесс Pj может войти в критическую секцию, только когда он получит ответные сообщения от всех остальных процессов. При этом сообщение с запросом от Pj достигло Pi после того, как Pi отправил свой запрос; в противном случае отметка времени запроса Pi должна была бы быть больше отметки времени запроса Pj по правилам работы логических часов.

По условиям алгоритма процесс Pi не может отослать ответное сообщение «REPLY» процессу Pj, находясь в состоянии запроса на вход в критическую секцию и имея меньшее значение отметки времени своего запроса, до тех пор, пока Pi не выйдет из критической секции.

Таким образом мы получаем противоречие с предположением, что Pj получил ответные сообщения REPLY от всех процессов в системе, в том числе, от процесса Pi.

Покажем теперь, что алгоритм Рикарта-Агравала обладает ***свойством живучести***. Процесс Pi не сможет получить доступ к критической секции, если он отправил свой запрос, но не получил необходимого ответа хотя бы от одного процесса Pj. В этом случае мы будем говорить, что Pi ждет Pj. Эта ситуация возможна, если процесс Pj находится либо в состоянии выполнения внутри критической секции, либо в состоянии запроса на вход в критическую секцию. При этом отметка времени запроса Pj должна быть меньше отметки времени запроса Pi.

Если Pj выполняется внутри критической секции, то через ограниченное время он выйдет из критической секции и отправит ответное сообщение процессу Pi.

Если же Pj находится в состоянии запроса на вход в критическую секцию, то либо он войдет в критическую секцию, либо должен существовать другой процесс Pk такой, что Pj ждет Pk.

Продолжая эти рассуждения, мы построим цепь из процессов, ожидающих друг друга: Pi ждет Pj, Pj ждет Pk, Pk ждет Pm, и т.д. Важно отметить, что эта цепь имеет ограниченную длину и является ациклической: процессы располагаются в ней в порядке убывания отметок времени запросов.

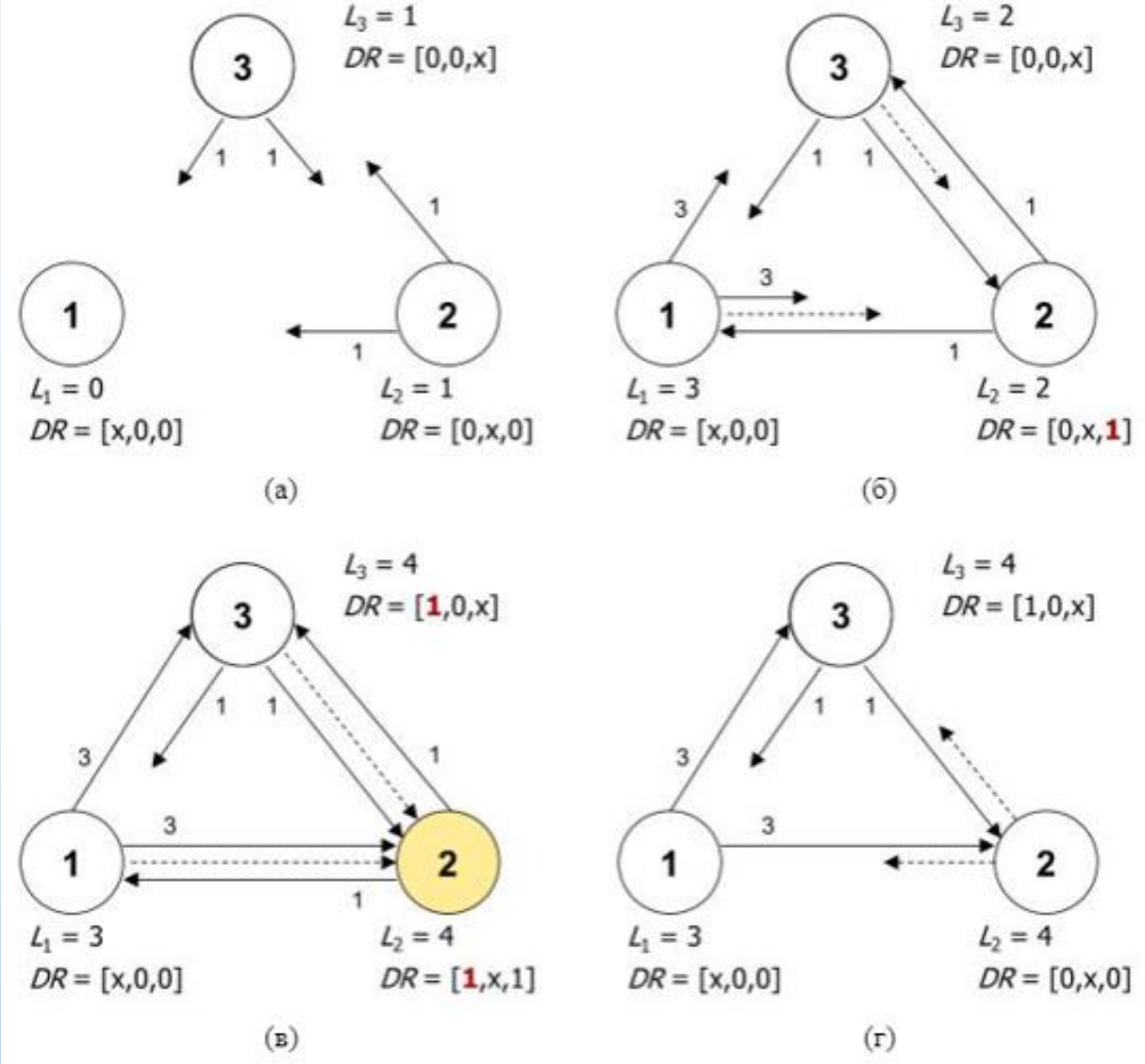
Последний процесс Pl в этой цепи с наименьшим временем запроса либо выполняется в критической секции, либо получит разрешения от всех остальных процессов и сможет войти в критическую секцию.

При выходе из критической секции Pl разошлет ответные сообщения всем ожидающим процессам, в том числе процессу, стоящему непосредственно слева от него, тем самым позволяя этому процессу войти в критическую секцию. Поэтому за ограниченное число шагов любой процесс окажется последним в цепи ожидающих процессов и сможет войти в критическую секцию.

Из этих рассуждений видно, что в алгоритме Рикарта-Агравала, также как и в алгоритме Лэмпорта, доступ к критической секции предоставляется строго в соответствии с отметками времени запросов по порядку, сохраняющему частичный причинно-следственный порядок →. Отсюда следует, что после получения сообщения REQUEST от процесса Pi всеми процессами ни один из них не сможет войти в критическую секцию более одного раза вперед Pi.

Обратим также внимание, что в отличие от алгоритма Лэмпорта, в котором запросы на доступ к критической секции явным образом выстраивались в очередь, в алгоритме Рикарта-Агравала процессы, желающие войти в критическую секцию, неявно упорядочиваются в ациклическую цепь, в которой каждый процесс ожидает поступления разрешений от процессов, стоящих справа от него.

Пример работы алгоритма Рикарта-Агравала приведен на рисунке 3.



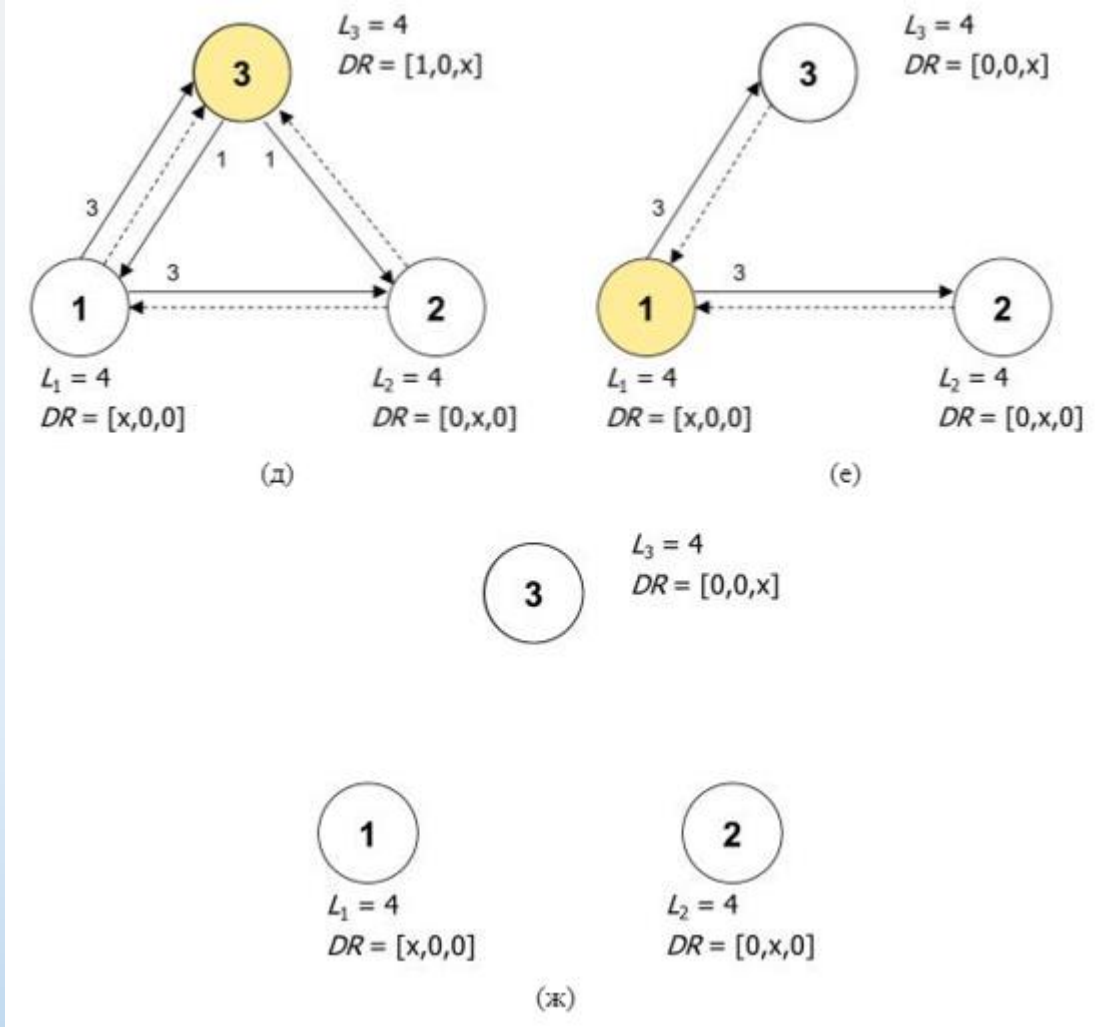


Рисунок 3 – Пример работы алгоритма Рикарта-Агравала

Также как и в сценарии, представленном выше для алгоритма Лэмпорта, на рисунке 3а процессы P2 и P3 запрашивают вход в критическую секцию приблизительно в одно и то же время.

На рисунке 3б процессы P1 и P3 получают запрос от P2 и отправляют ему ответные сообщения. В свою очередь процесс P2, получив запрос от P3, откладывает отправку ответного сообщения, так как P2 находится в состоянии запроса на вход в критическую секцию и отметка времени его запроса меньше отметки времени запроса P3. Соответствующий элемент массива DR2 устанавливается в единицу. Поэтому процесс P3 не сможет получить доступ к критической секции до тех пор, пока P2 не войдет и не выйдет из нее.

Далее в нашем сценарии процесс P1 переходит в состояние запроса на вход в критическую секцию и рассылает соответствующее сообщение остальным процессам, как показано на рисунке 3б.

Между тем, получив необходимые ответы от P1 и P3, процесс P2 входит в критическую секцию (рисунок 3в). Затем запрос от процесса P1 достигает процессов P2 и P3. Обратим внимание, что при получении запроса от P1 процессы P2 и P3 не отправляют ответные сообщения, так как P2 находится в состоянии выполнения в критической секции, а P3 находится в состоянии запроса на вход в критическую секцию и отметка времени его запроса меньше отметки времени запроса P1.

В итоге цепь ожидающих процессов будет выглядеть следующим образом: P1 ждет P3, P3 ждет P2. Эта ситуация проиллюстрирована на рисунке 3в.

Действительно, процесс P3 знает о том, что его ждет процесс P1, так как DR3[1] = 1, а процесс P2 знает о том, что его ждут процессы P1 и P3, так как DR2[1] = 1 и DR2[3] = 1.

На рисунке 3г процесс P2 выходит из критической секции и рассылает отложенные ответы ожидающим процессам P1 и P3. Теперь последним в цепи ожидающих процессов оказывается процесс P3, и он войдет в критическую секцию следующим.

На рисунке 3д запрос процесса P3 достигает процесса P1, и несмотря на то, что сам P1 находится в состоянии запроса на вход в критическую секцию, он высылает P3 ответ, так как отметка времени запроса P1 больше отметки времени запроса P3.

По получению необходимых ответов P3 входит в критическую секцию. Когда P3 выйдет из критической секции, он отправит P1 отложенный ответ, и P1 сможет войти в критическую секцию, как показано на рисунке 3е.

На рисунке 3ж процесс P1 выходит из критической секции. При этом ему не требуется рассылать никаких сообщений.

Отметим, что в отличие от алгоритма Лэмпорта, алгоритм РикартаАгравала использует для работы с критической секцией 2(N – 1) сообщений: (N – 1) сообщений требуется для входа в критическую секцию и (N – 1) сообщений требуется для выхода из критической секции.

4. Алгоритмы на основе передачи маркера

Для алгоритмов данного класса право войти в критическую секцию 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 критическую секцию до тех пор, пока маркер не будет передан другому процессу. Очевидно, что в течение всего времени выполнения внутри критической секции 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 критическую секцию.

Самым простым решением, обеспечивающим такие гарантии, является организация ***непрерывного перемещения маркера среди всех процессов распределенной системы (perpetuum mobile)***. Чтобы не пропустить ни одного процесса, желающего войти в критическую секцию, 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 (token ring)***.

Любой процесс, желающий войти в критическую секцию, ожидает прихода маркера, и после выхода из критической секции передает его дальше по кольцу. Если процесс, получивший маркер, не заинтересован в работе с критической секцией, он просто передает маркер своему соседу. Поэтому даже если ни один из процессов не работает с критической секцией, маркер продолжает непрерывно циркулировать между процессами. Поскольку маркер перемещается от процесса к процессу в общеизвестном порядке, ситуации голодания возникнуть не может. Когда процесс решает войти в критическую секцию, в худшем случае ему придется ожидать, пока все остальные процессы последовательно войдут в критическую секцию и выйдут из нее.

Рассмотрим алгоритмы, в которых перемещение маркера осуществляется лишь в ответ на получение запроса на владение маркером от заинтересованного в нем процесса (token asking methods). Отметим, что в этом случае необходимо создать такие условия распространения запросов, при которых каждый запрос рано или поздно достигнет процесса, в котором находится маркер, вне зависимости от перемещений самого маркера.

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 критическую секцию, 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 критической секции, он откладывает передачу маркера до тех пор, пока не выйдет из нее.

Отметим, что для работы алгоритма Сузуки-Касами не требуется, чтобы каналы связи обладали свойством FIFO.

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

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. Более того, из-за различных и меняющихся задержек передачи сообщений, процесс может получить запрос уже после того, как этот запрос был удовлетворен, то есть после того, как процесс, рассылавший этот запрос, уже получал маркер в свое владение. 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 критической секции, он должен определить список процессов, находящихся в состоянии запроса на вход в критическую секцию, для того, чтобы передать маркер одному из них.

Для решения этих задач в алгоритме Сузуки-Касами каждый процесс использует ***порядковые номера (sequence numbers)*** запросов на вход в критическую секцию. Порядковый номер n (n = 1, 2, …) увеличивается процессом Pi независимо от других процессов каждый раз, когда Pi формирует новый запрос на вход в критическую секцию. 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] (request number), 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).

Сам маркер переносит внутри себя очередь Q идентификаторов процессов, находящихся в состоянии запроса на вход в критическую секцию, и массив LN[1..N] (last number), в элементе LN[i] которого содержится порядковый номер критической секции, в которой в последний раз выполнялся процесс Pi. Чтобы поддерживать элементы массива LN[1..N] в актуальном состоянии, при выходе из критической секции процесс Pi обновляет элемент LN[i] = RNi[i], тем самым указывая, что его запрос с порядковым номером RNi[i] был удовлетворен.

Если все процессы будут следовать этому правилу, то находящийся у Pi запрос процесса Pj с порядковым номером RNi[j] будет устаревшим, если RNi[j] ≤ LN[j]. Если же RNi[j] = LN[j] + 1, то это значит, что процесс Pj ожидает получения маркера, и запрос Pj с порядковым номером RNi[j] еще не был обслужен. В этом случае процесс Pi, владея маркером, помещает идентификатор процесса Pj в конец очереди Q при условии, что идентификатор Pj еще не присутствует в Q. После этого Pi передает маркер процессу, идентификатор которого является первым в очереди Q.

Таким образом, алгоритм Сузуки-Касами будет определяться ***следующими правилами***:

Запрос на вход в критическую секцию:

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

Вход в критическую секцию:

Процесс Pi может войти в критическую секцию, если он обладает маркером.

Выход из критической секции:

При выходе из критической секции процесс Pi 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 Pi добавляет идентификатор Pj в конец очереди Q.

Если после выполнения представленных выше операций с очередью Q она оказывается не пустой, процесс Pi 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 Pi на вход в критическую секцию 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 Pi 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 Pi будет обслужен, и Pi получит маркер.

Пример работы алгоритма Сузуки-Касами приведен на рисунке 4.

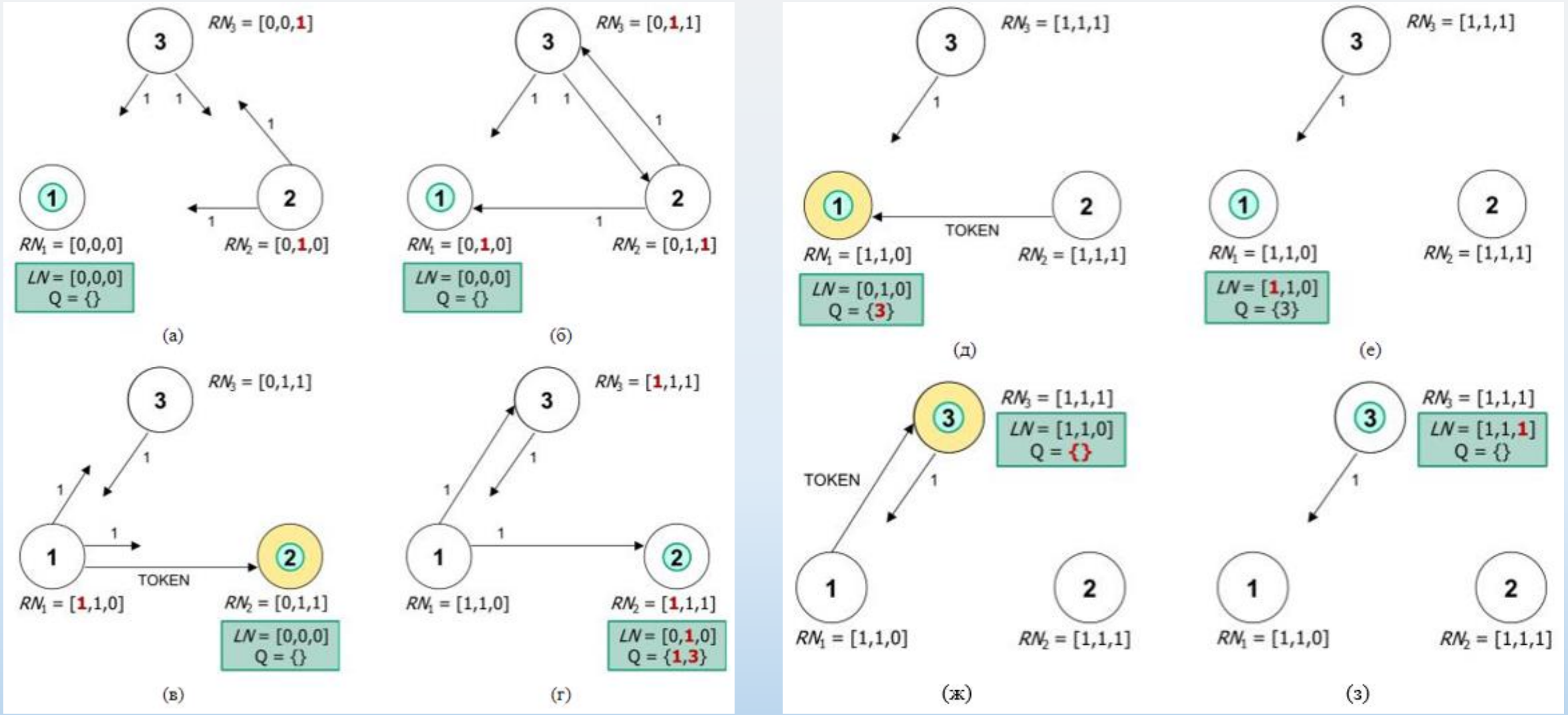


Рисунок 4 – Пример работы широковещательного алгоритма Сузуки-Касами

В начальном состоянии системы маркером владеет процесс P1, и все процессы находятся в состоянии выполнения вне критической секции. Также как и в сценарии, представленном в примерах выше, процессы P2 и P3 запрашивают вход в критическую секцию приблизительно в одно и то же время, как показано на рисунке 4а.

На рисунке 4б в процесс P1 первым поступает запрос от процесса P2, поэтому маркер будет передан именно P2. Обратим внимание, что если бы в P1 первым поступил запрос от процесса P3, то маркер был бы отправлен P3. Такая зависимость порядка входа в критическую секцию от меняющихся задержек передачи сообщений отличает алгоритм Сузуки-Касами от алгоритмов Лэмпорта и Рикарта-Агравала, предоставляющих доступ к критической секции строго в соответствии с отметками времени запросов и независимо от задержек передачи сообщений.

На рисунке 4в процесс P2 получает маркер от P1 в виде сообщения TOKEN и входит в критическую секцию. Далее в нашем сценарии процесс P1 переходит в состояние запроса на вход в критическую секцию и рассылает соответствующее сообщение остальным процессам.

Запрос процесса P1 достигает процессов P2 и P3, как показано на рисунке 4г. После этого процесс P2 выходит из критической секции и обновляет значение LN[2] и содержимое очереди Q. А именно, значение элемента LN[2] меняется на единицу, тем самым, указывая, что процесс P2 уже выполнялся в своей критической секции с порядковым номером один, а в очередь Q добавляются идентификаторы процессов P1 и P3, ожидающих получения маркера.

Важно отметить, что порядок добавления идентификаторов процессов в очередь Q и, как следствие, порядок обслуживания запросов на вход в критическую секцию, определяется порядком просмотра массива запросов RNi. В нашем примере процесс P2 при выходе из критической секции просматривает RN2 с начала, поэтому идентификатор процесса P1 будет помещен в очередь Q вперед идентификатора P3 даже несмотря на то, что P1 запросил доступ к критической секции после P3.

Если бы при выходе из критической секции процесс Pi просматривал массив RNi начиная с позиции i + 1 и до N, а затем с позиции 1 до i – 1, то для нашего примера процесс P2 поместил бы идентификатор P3 перед идентификатором P1.

На рисунке 4д процесс P2 передает маркер процессу, находящемуся во главе очереди Q, то есть процессу P1, и процесс P1 входит в критическую секцию. Интересно отметить, что на рисунке 4д процесс P1 знает, что процесс P3 ожидает маркер, хотя P1 так и не получал запроса от P3: сообщение с запросом на владение маркером от P3 еще не поступило в P1. Эта информация доступна P1 из состояния очереди Q, которую он получил вместе с маркером, и в которую эти сведения были добавлены ранее процессом P2. Поэтому при выходе из критической секции P1 передаст маркер процессу P3, так как в очереди Q содержится только идентификатор P3.

На рисунках 4е и 4ж процесс P3 наконец получит маркер и войдет в критическую секцию. При выходе из критической секции P3 обновит значение LN[3] и сохранит маркер у себя, так как в системе нет необслуженных запросов, как показано на рисунке 4з. В свою очередь запрос процесса P3 с порядковым номером n = 1, который рано или поздно будет получен процессом P1, будет рассматриваться как устаревший, так как RN1[3] = 1 и LN[3] = 1.

В заключение отметим, что если процесс не владеет маркером, то для входа в критическую секцию 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 критическую секцию ему не потребуется ни одного сообщения.

Алгоритм Реймонда на основе покрывающего дерева. Для уменьшения числа сообщений, обмен которыми требуется для работы с критической секцией, в алгоритме Реймонда предполагается, что все процессы распределенной системы логически организованы в дерево таким образом, что сообщения пересылаются только вдоль его ненаправленных ребер, каждое из которых соответствует паре разнонаправленных каналов связи между процессами. Такое дерево может представлять собой минимальное покрывающее дерево графа, описывающего физическую топологию сетевых соединений, или может быть построено для сети с полносвязной логической топологией, используемой в алгоритме Сузуки-Касами. Отметим, что между любыми двумя вершинами в дереве, например, между вершинами, соответствующими процессу, запрашивающему маркер, и процессу, владеющему маркером, существует ровно один путь, вдоль которого эти процессы обмениваются сообщениями REQUEST и TOKEN.

Благодаря такой организации процессов каждый процесс имеет представление только о своих непосредственных соседях и взаимодействует только с ними, что отличает данный алгоритм от широковещательного алгоритма Сузуки-Касами.

Например, на рисунке 5 процесс P1 поддерживает связь только с процессами P2, P3 и P4 и может даже не знать о существовании процессов P5 и P6.

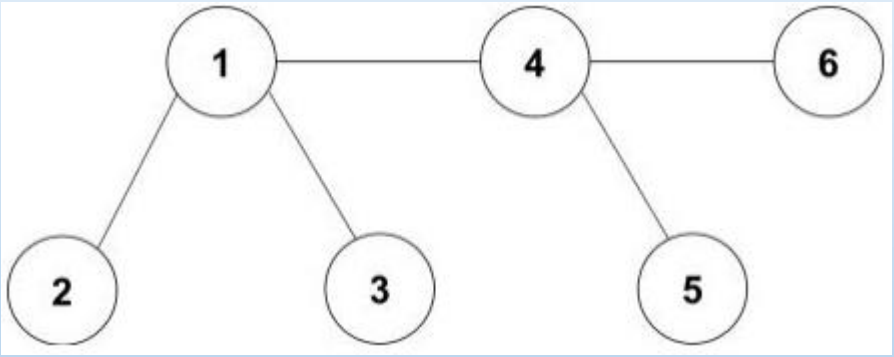


Рисунок 5 – Пример дерева процессов

Для того чтобы каждый запрос на владение маркером смог достичь процесса, в котором находится маркер, каждый процесс в дереве поддерживает работу с переменной Holder, указывающей на расположение маркера относительно этого процесса. А именно, не владеющий маркером процесс Pi в своей переменной Holderi хранит идентификатор своего соседнего процесса Pj, который, по его мнению, обладает маркером или через который проходит путь в процесс, владеющий маркером. Поэтому относительно Pi такой соседний процесс Pj будет являться корнем поддерева, в одной из вершин которого содержится маркер. Если же Pi сам владеет маркером, то переменная Holderi устанавливается в значение self.

Ситуация, когда относительно Pi маркер находится в поддереве, корнем которого является Pj, то есть когда Holderi = j, графически задается путем введения направления для ребра между вершинами Pi и Pj в сторону от Pi к Pj. Таким образом, объединяя информацию, хранящуюся в переменных Holder различных процессов, можно построить единственный направленный путь от любого процесса, не владеющего маркером, к процессу с маркером, что проиллюстрировано на рисунке 6 для случая, когда маркер содержится в процессе P6.

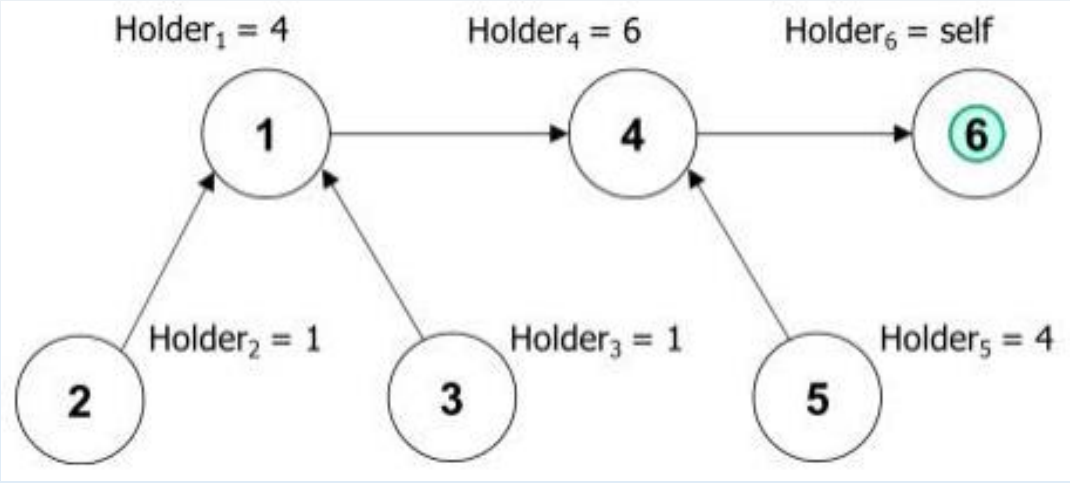


Рисунок 6 – Пример дерева процессов с ребрами, направленными в сторону процесса с маркером

Если не владеющий маркером процесс Pi собирается войти в критическую секцию, он отправляет сообщение REQUEST с запросом на получение маркера соседнему процессу Pj, идентификатор которого содержится в переменной Holderi. В свою очередь, если Pj не владеет маркером, при получении сообщения REQUEST от Pi он пересылает этот запрос дальше в сторону процесса, владеющего маркером, то есть процессу, на который указывает значение его переменной Holderj. Таким образом, сообщения REQUEST последовательно проходят вдоль направленного пути от процесса, запрашивающего маркер, к процессу, обладающему маркером.

К примеру, если процесс P1 на рисунке 6 захочет войти в критическую секцию, он отправит запрос процессу P4, так как Holder1 = 4. В свою очередь благодаря тому, что Holder4 = 6, процесс P4 переправит запрос процессу P6, у которого и находится маркер.

Процесс, владеющий маркером и находящийся в состоянии выполнения вне критической секции, должен передать маркер одному из своих соседей, от которых он получал сообщение REQUEST. Для рассматриваемого примера, процесс P6 передаст маркер процессу P4, и изменит значение своей переменной Holder6 = 4. Процесс P4 не запрашивал маркер для себя, а отправлял сообщение REQUEST «по просьбе» процесса P1, поэтому P4 должен переслать маркер процессу P1, устанавливая при этом Holder4 = 1.

При получении маркера процесс P1 установит Holder1 в значение self и получит возможность войти в критическую секцию, как показано на рисунке 7.

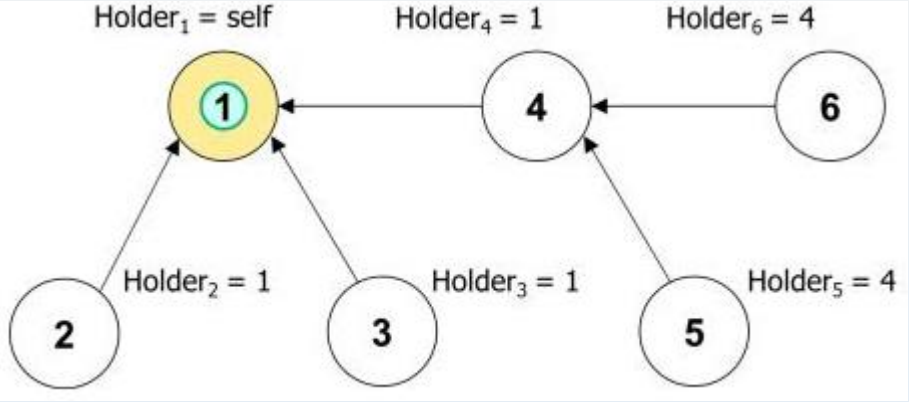


Рисунок 7 – Пример перемещения маркера

Обратим внимание, что при пересылке маркера переменные Holder изменяют свои значения таким образом, что совместно эти значения всегда определяют направленный путь от любого процесса в системе к процессу, владеющему маркером.

Отметим, что описанный в общих чертах алгоритм, строго говоря, не является «полностью распределенным» в отличии, например, от алгоритма Рикарта-Агравала, в котором все процессы в равной степени принимают участие в решении о предоставления доступа к критической секции.

Дело в том, что согласно такому определению любой «полностью распределенный» алгоритм всегда будет требовать обмен O(N) сообщениями для входа в критическую секцию. В представленном же алгоритме процесс Pi просит соседний процесс Pj запрашивать маркер от имени Pi. Это позволяет процессу Pj выступать от лица всех своих соседей, тем самым уменьшая число сообщений, требуемых для работы с критической секцией.

Для реализации алгоритма Реймонда каждый процесс Pi поддерживает работу со следующими переменными и структурами данных:

1. Переменная Holder. Возможные значения: self или идентификатор одного из непосредственных соседей процесса Pi. Указывает на расположение маркера по отношению к данному процессу.

2. Переменная Using. Возможные значения: true или false. Служит признаком того, выполняется ли процесс Pi в критическую секцию или нет. Очевидно, что если Using = true, то Holder = self.

3. Очередь Q. Возможные значения: self или идентификатор одного из непосредственных соседей процесса Pi. Очередь Q содержит идентификаторы процессов, от которых Pi получал сообщение REQUEST с запросом на обладание маркером, и которые еще не получали маркер в ответ. Значение self помещается в очередь Q, когда Pi сам собирается войти в критическую секцию, то есть нуждается в маркере. В связи с тем, что каждый идентификатор помещается в очередь Q не более одного раза, максимальный размер очереди Q равен числу соседей процесса Pi плюс один для значения self.

4. Переменная Asked. Возможные значения: true или false. Переменная Asked содержит значение true, если не обладающий маркером процесс Pi уже отправлял сообщение REQUEST процессу, на который указывает Holderi; в противном случае Asked = false. Переменная Asked предотвращает отправку излишних сообщений REQUEST, а также гарантирует отсутствие дубликатов в очереди Q.

Для работы с КС каждый процесс должен реализовать функцию `AssignPrivilege()`, с помощью которой осуществляется передача маркера другому процессу или использование маркера для входа в критическую секцию, и функцию `MakeRequest()`, с помощью которой отправляется запрос на получение маркера. Пример кода для этих функций приведен в Листинге 2 и Листинге 3.

Листинг 2. Функция работы с маркером.

```
void AssignPrivilege()
{
/* Функция работы с маркером */
if (Holder == self && !Using && !Q.empty())
{
/* Выборка первого элемента из очереди Q */
Holder = Q.front(); Q.pop();
Asked = false;
/* Если первый элемент self, процесс входит в КС */
/* иначе передает маркер процессу с ID в Holder */
if (Holder == self)
{
Using = true;
/* вход в КС */
}
else
{
/* отправка TOKEN процессу с ID в Holder */
}
}
}
```

Листинг 3. Функция отправки запроса на получение маркера.

```
void MakeRequest()
{
/* Функция отправки запроса на получение маркера */
if (Holder != self && !Asked && !Q.empty())
{
/* отправка REQUEST процессу с ID в Holder */
Asked = true;
}
}
```

Функция работы с маркером. Отправлять маркер другому процессу или использовать его для входа в критическую секцию может только процесс, для которого одновременно выполняются следующие условия:

(1) Holder = self,

(2) Using = false, (3) очередь Q не пуста.

Если значение self не является первым элементом в очереди Q, то процесс, владеющий маркером, пересылает его соседнему процессу, идентификатор которого находится во главе очереди Q. При этом он удаляет этот элемент из Q, т.к. передача маркера по соответствующему запросу становится выполненной, и обновляет значение своей переменной Holder таким образом, чтобы оно указывало на нового обладателя маркера. Ситуация, когда в очереди Q процесса, владеющего маркером, первым элементом является self, возможна, например, когда этот процесс только что получил сообщение TOKEN от своего соседа. В этом случае он получает право войти в критическую секцию, удалив при этом первый элемент со значением self из начала очереди, и присвоив переменной Using значение true. Переменная Asked устанавливается в значение false, так как при передаче маркера процесс может не отправлять сообщение REQUEST процессу, на который указывает новое значение его переменной Holder.

Функция отправки запроса на получение маркера. Процесс, не владеющий маркером, отправляет сообщение REQUEST, если он нуждается в нем для себя или для выполнения просьбы на получение маркера от своего соседа, то есть, когда одновременно выполняются следующие условия:

(1) Holder не равно self, (2) очередь Q не пуста,

(3) Asked = false.

Проверка переменной Asked нужна для предотвращения отправки повторных сообщений REQUEST процессу, на который указывает Holder. Поэтому при отправке запроса REQUEST переменная Asked устанавливается в значение true. Важно отметить, что отправка сообщения REQUEST не меняет значения других переменных и структур данных, поддерживаемых процессом: значения переменных Holder, Using и очередь Q остаются прежними.

Алгоритм Реймонда опирается на представленные выше функции `AssignPrivilege` и `MakeRequest` и состоит из четырех частей, соответствующих четырем возможным событиям, наступающим в каждом процессе:

Запрос на вход в критическую секцию:

Процесс Pi, желающий войти в КС, помещает значение self в свою очередь Q и выполняет функцию `AssignPrivilege` и, затем, `MakeRequest`. Если Pi владеет маркером, функция `AssignPrivilege` либо предоставит возможность процессу войти в КС, либо передаст маркер другому процессу. Если маркер не содержится в Pi, функция `MakeRequest` может направить соответствующий запрос на получение маркера.

Получение сообщения REQUEST:

При получении сообщения REQUEST от процесса Pj процесс Pi помещает идентификатор Pj в свою очередь Q и выполняет функцию `AssignPrivilege` и, затем, `MakeRequest`. Если Pi владеет маркером, с помощью функции `AssignPrivilege` процесс Pi может отослать маркер запрашивающему процессу. Если маркер не содержится в Pi, с помощью функции `MakeRequest` Pi может передать запрос от Pj в сторону процесса с маркером.

Получение сообщения TOKEN:

При получении сообщения TOKEN процесс Pi изменяет значение Holder на self и выполняет функцию `AssignPrivilege` и, затем, `MakeRequest`. С помощью функции `AssignPrivilege` процесс Pi может передать маркер другому процессу или осуществить вход в критическую секцию. Если маркер будет передан другому процессу, с помощью функции `MakeRequest` процесс Pi может запросить возврат маркера себе.

Выход из критической секции:

При выходе из КС процесс Pi изменяет значение Using на false и выполняет функцию `AssignPrivilege` и, затем, `MakeRequest`. С помощью функции `AssignPrivilege` процесс Pi может передать маркер другому процессу, а выполнение функции `MakeRequest` позволит запросить возврат маркера обратно в Pi.

Обратим внимание, что процесс может войти в критическую секцию, если он обладает маркером, и при этом первый элемент в его очереди Q имеет значение self. Эта ситуация проиллюстрирована в коде функции `AssignPrivilege`, представленном в Листинге 2.

Благодаря организации процессов в логическое дерево возникновение конфликтных ситуаций, связанных с произвольными задержками передачи сообщений и изменением порядка их поступления в каналах, не обладающих свойством FIFO, ограничено взаимодействием двух соседних процессов. Поэтому в алгоритме Реймонда нет необходимости использовать порядковые номера (англ. sequence numbers) сообщений. Алгоритм работает таким образом, что обмен сообщениями между двумя соседними процессами подчиняется определенному шаблону взаимодействия, проиллюстрированному на рисунке 8.

Рисунок 8 – Шаблон взаимодействия соседних процессов

Предполагается, что изначально маркер находится в одной из вершин поддерева, корнем которого является процесс Pi.

Как следует из рисунка 8, единственно возможной ситуацией нарушения порядка поступления сообщений является ситуация, когда отправленное процессом Pi сообщение REQUEST, следующее за сообщением TOKEN, будет получено процессом Pj вперед сообщения TOKEN. На самом деле, процесс Pj в состоянии распознать такую ситуацию, так как согласно представленному шаблону взаимодействия, следующим ожидаемым сообщением должно быть именно сообщение TOKEN, и Pj сможет отложить обработку сообщения REQUEST до получения TOKEN.

Однако это не является необходимым, так как такое нарушение порядка поступления сообщений не оказывает влияния на функционирование алгоритма. Действительно, если REQUEST поступает в процесс Pj раньше, чем TOKEN, то идентификатор процесса Pi будет помещен в конец очереди Qj процесса Pj. В связи с тем, что Pj пока еще не владеет маркером, функция `AssignPrivilege` выполнена не будет. Факт того, что Pi отправил маркер Pj, означает, что очередь Qj была не пуста, и переменная Askedj процесса Pj установлена в значение true, и поэтому функция `MakeRequest` также не будет выполнена, что не позволит запросу REQUEST распространиться среди других процессов.

Когда сообщение TOKEN, в конце концов, доберется до процесса Pj, он войдет в критическую секцию или передаст маркер соседнему процессу, идентификатор которого находится в начале его очереди Qj. Этим процессом не может быть процесс Pi, передающий маркер Pj, так как его запрос был добавлен в конец непустой очереди Qj. Поэтому получение процессом Pj сообщения REQUEST вперед сообщения TOKEN не оказывает никакого влияния на его работу.

Начальное состояние системы и инициализация алгоритма. В начальном состоянии распределенной системы маркер должен находиться только у одного процесса. Процесс, обладающий маркером, устанавливает переменную Holder в значение self и отправляет сообщение INITIALIZE всем своим непосредственным соседям. При получении сообщения INITIALIZE процесс Pi присваивает переменной Holderi значение идентификатора процесса, от которого он получил INITIALIZE, и рассылает INITIALIZE далее всем своим соседним процессам за исключением процесса, от которого было получено сообщение INITIALIZE. Как только процесс получает сообщение INITIALIZE, он может запрашивать маркер, даже если еще не все процессы в дереве прошли процедуру инициализации. Начальное значение переменных Using и Asked равно false для всех процессов, очередь Q всех процессов пуста.

Теперь покажем, что алгоритм Реймонда обладает свойством живучести, то есть не допускает ситуаций взаимной блокировки и голодания.

При рассмотрении возможности возникновения взаимной блокировки предположим, что допустима ситуация, когда ни один процесс не выполняется в критической секции, и существует один или несколько процессов, находящихся в состоянии запроса на вход в критическую секцию, но не получающих разрешения войти в нее. Такая ситуация возможна только при реализации одного из следующих сценариев:

- ни один из процессов не владеет маркером, поэтому маркер не может быть передан процессу, ожидающему его получения;
- обладающий маркером процесс не имеет информации, что другой процесс нуждается в маркере;
- сообщение TOKEN не может достигнуть процесса, запросившего доступ к критической секции.

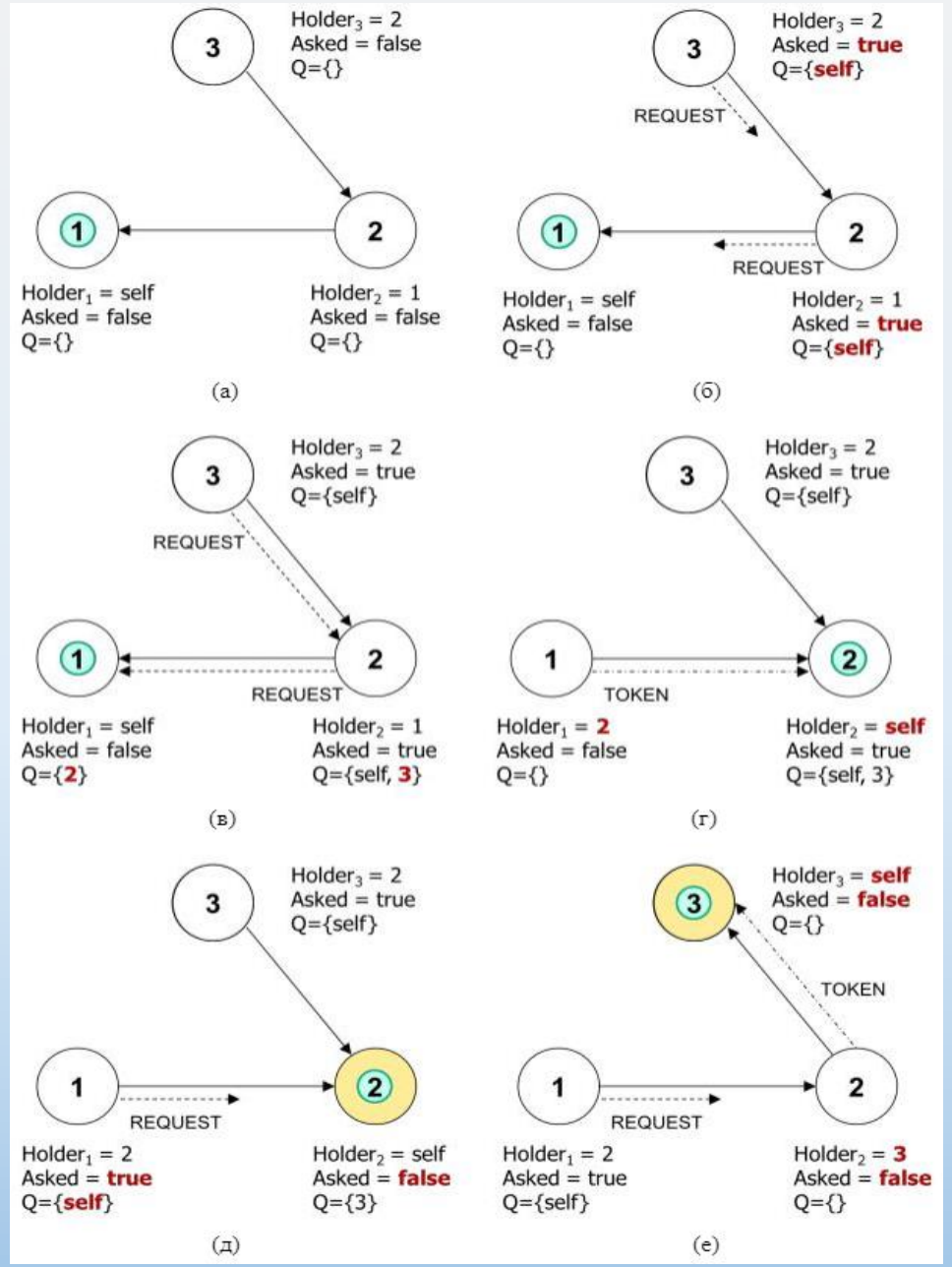
Первый сценарий невозможен ввиду предположений, что каналы связи являются надежными, а процессы не выходят из строя.

Совокупность переменных Asked различных процессов гарантирует, что существует последовательность сообщений REQUEST, отправленных вдоль пути, сформированному переменными Holder, от процесса, запросившего доступ к критической секции, к процессу, обладающему маркеру. Ситуация, когда сообщение TOKEN перемещается по дереву таким образом, что сообщение REQUEST никогда не достигнет процесса, в котором находится маркер, невозможна ввиду отсутствия циклов в дереве процессов. Единственной возможностью, при которой сообщение TOKEN может разминуться с сообщением REQUEST, является передача TOKEN от процесса Pi к соседнему процессу Pj в то время, как REQUEST передается в обратном направлении, то есть от Pj к Pi. Однако шаблон взаимодействия соседних процессов, представленный на рисунке 8, предотвращает такую ситуацию. А именно, сообщение TOKEN может быть отослано процессом Pi только после получения от Pj запроса REQUEST, на который Pi еще не отвечал. Если же Pj уже отправлял такой запрос процессу Pi, то тогда Pj не станет отправлять еще один запрос REQUEST, так как значение его переменной Asked будет равно true.

Таким образом процесс, владеющий маркером, рано или поздно получит запрос от соседнего с ним процесса, нуждающегося в маркере для себя или для выполнения просьбы на получение маркера от другого процесса. Идентификаторы процессов, отправляющих сообщения REQUEST вдоль пути, ведущего к процессу с маркером, помещаются в очереди Q промежуточных процессов таким образом, что, совместно, эти идентификаторы формируют обратный логический путь для передачи маркера к запрашивающему процессу. Поэтому, следуя этому пути, сообщение TOKEN может быть доставлено процессу, запросившему вход в критическую секцию.

Голодание в алгоритме Реймонда также невозможно, так как очереди Q промежуточных процессов на пути от запрашивающего процесса к процессу с маркером имеют ограниченный размер и обслуживаются по правилу FIFO. Поэтому каждый из промежуточных процессов рано или поздно (в зависимости от позиции его идентификатора в очереди Q соседнего промежуточного процесса) получит маркер и сможет передать его по обратному пути в сторону запрашивающего процесса. Таким образом, как только сообщение REQUEST от запрашивающего процесса достигнет процесса, владеющего маркером, сообщение TOKEN, в конце концов, гарантировано доберется до процесса, желающего войти в критическую секцию.

Пример работы алгоритма Реймонда приведен на рисунке 9.



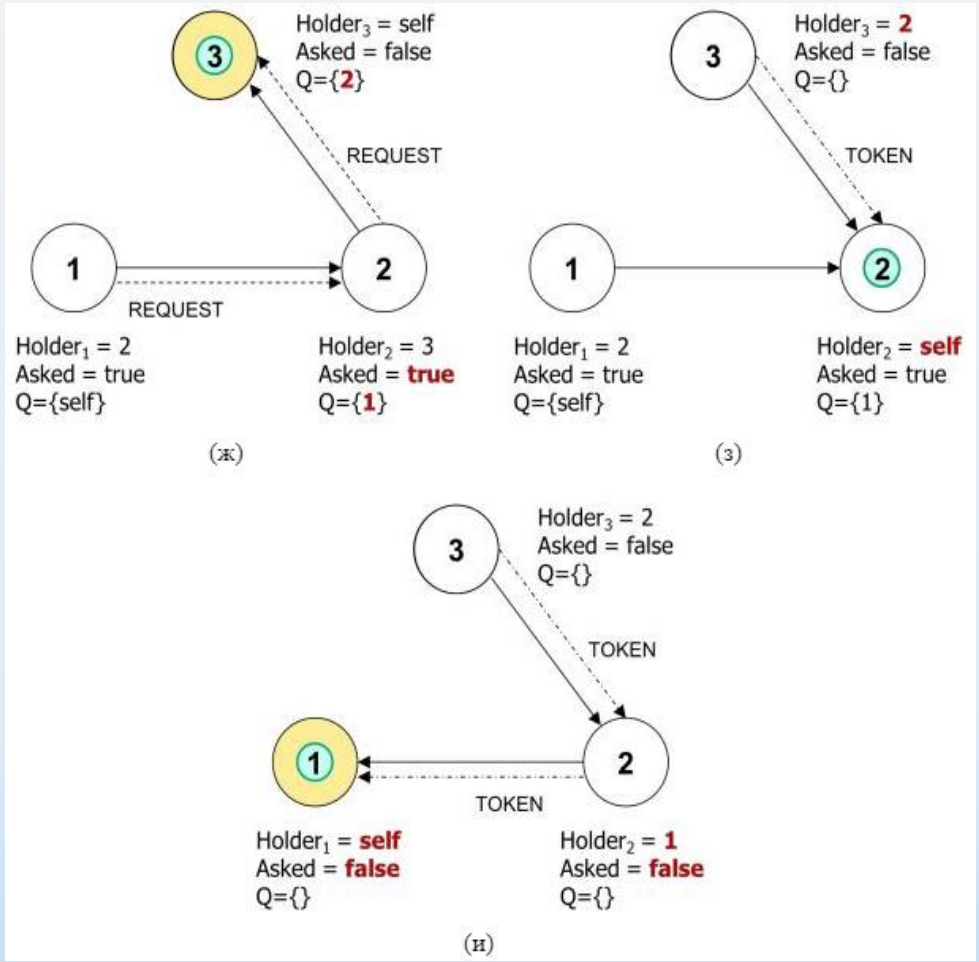


Рисунок 9 – Пример работы алгоритма Реймонда

В начальном состоянии системы маркером владеет процесс P1, и все процессы находятся в состоянии выполнения вне критической секции, как показано на рисунке 9а. Сетевые соединения установлены только между процессами P1 и P2, P2 и P3. Направление ребер в дереве процессов обозначено стрелками, указывающими на расположение маркера относительно каждого процесса.

На рисунке 9б, также как и в сценарии, представленном в примерах выше, процессы P2 и P3 запрашивают вход в КС приблизительно в одно и то же время, и каждый из них отправляет сообщение REQUEST своему соседу, идентификатор которого содержится в переменной Holder. При получении запроса от P3 процесс P2 не передает его P1, так как Asked2 = true (рисунок 9в).

При этом сообщение REQUEST, отправленное ранее процессом P2 процессу P1, начинает представлять интересы и P2, и P3.

На рисунке 9г процесс P1 передает маркер процессу P2, одновременно изменяя значение своей переменной Holder так, чтобы оно указывало на P2, как на нового обладателя маркера. В начале очереди Q процесса P2 стоит self, поэтому при получении маркера этот процесс сможет войти в критическую секцию, как показано на рисунке 9д.

Далее в нашем сценарии процесс P1 переходит в состояние запроса на вход в критическую секцию и отправляет сообщение REQUEST процессу P2.

На рисунке 9е процесс P2 выходит из критической секции и передает маркер процессу, идентификатор которого содержится во главе его очереди Q, то есть процессу P3. Получив маркер, процесс P3 войдет в критическую секцию.

После этого запрос процесса P1 достигает процесса P2, и P2 передает запрос процессу P3, как показано на рисунке 9ж. Обратим внимание, что идентификаторы процессов P3 и P2, находящиеся в очередях Q3 и Q2, совместно образуют логический путь для передачи маркера из процесса P3, владеющего маркером, в процесс P1, запрашивающий маркер.

При выходе процесса P3 из критической секции сообщение TOKEN будет отправлено по этому пути: сначала в P2 и затем в P1, как показано на рисунке 9з и рисунке 9и. Также с перемещением маркера будут изменяться значения переменных Holder таким образом, что совместно эти значения всегда будут указывать на текущее положение маркера. В итоге, получив маркер, процесс P1 сможет войти в критическую секцию.

Чтобы оценить эффективность работы алгоритма Реймонда отметим, что наибольшее число сообщений, требуемых для входа в критическую секцию, составляет 2D, где D – диаметр дерева процессов, то есть максимальное из расстояний между парами вершин дерева. Эта ситуация возникает для случая наибольшего удаления запрашивающего процесса от процесса с маркером, и требует D сообщений REQUEST и D сообщений TOKEN для входа в критическую секцию.

Поэтому самой неудачной топологией соединений между процессами является прямая линия, когда длина пути между крайними процессами составляет N – 1 ребер.

В этом случае при использовании алгоритма Реймонда может потребоваться обмен 2(N – 1) сообщениями, что совпадает с числом сообщений, используемых в алгоритме Рикарта-Агравала, и превышает число сообщений в алгоритме Сузуки-Касами. Такая ситуация соответствует постоянной передаче маркера из одного крайнего процесса в другой. В случае, когда каждый из процессов может запросить вход в критическую секцию с одинаковой вероятностью, среднее расстояние между запрашивающим процессом и процессом с маркером составит (N + 1)/3 ребер при условии, что маркер не находится в процессе, желающем войти в критическую секцию. Тогда среднее число сообщений будет приблизительно составлять 2N/3, что меньше N сообщений, требуемых в алгоритме Сузуки-Касами. Если же маркер находится у запрашивающего процесса, для входа в критическую секцию не потребуется ни одного сообщения.

С точки зрения уменьшения числа сообщений, требуемых для входа в критическую секцию, предпочтительной топологией соединений между процессами является дерево с большой степенью ветвления K. В этом случае диаметр дерева и, следовательно, максимальное число сообщений будет определяться как $O(\log_{K-1}N)$. Однако важно отметить, что с ростом степени ветвления K также растет максимальный размер очереди Q, который определяется числом соседей каждого процесса.

При высокой интенсивности поступления запросов на вход в критическую секцию алгоритм Реймонда проявляет интересное свойство: с увеличением числа процессов, ожидающих маркер, число сообщений, требуемых для работы с критической секцией, уменьшается. Связано это с тем, что сообщение REQUEST, отправленное запрашивающим процессом, в этом случае обычно не проходит весь путь к процессу с маркером, а поступает в промежуточный процесс Pi, у которого переменная Askedi = true. При этом сообщение REQUEST, отправленное ранее процессом Pi в сторону Holderi, начинает представлять интересы всех процессов, для которых путь в процесс с маркером проходит через Pi. Далее, при получении маркера от процесса Pj процесс Pi и все его соседи смогут воспользоваться маркером до его возврата в Pj. Другими словами, маркер будет перемещаться внутри поддерева, корнем которого является Pi, до тех пор, пока не будут обслужены все запросы на вход в критическую секцию, появившиеся до момента поступления маркера в Pi (точнее, до поступления в Pi следующего за маркером запроса от Pj). Поэтому среднее расстояние, проходимое сообщением TOKEN между процессами, ожидающими вход в критическую секцию, будет много меньше диаметра дерева.

Если все процессы распределенной системы постоянно запрашивают вход в критическую секцию, сообщение TOKEN проходит все N – 1 ребер дерева ровно два раза (в прямом и обратном направлении) для предоставления доступа к критической секции каждому из N процессов. В связи с тем, что сообщение TOKEN отправляется в ответ на сообщение REQUEST, всего в системе будет передано 4(N – 1) сообщений. Поэтому при полной загрузке среднее число сообщений, требуемых для входа в критическую секцию, будет равняться $4(N – 1)/N \approx 4$.

Кольцевой алгоритм с маркером. Все процессы системы образуют логическое кольцо, то есть каждый процесс знает номер своей позиции в кольце, а также номер ближайшего к нему следующего процесса (рисунок 10).

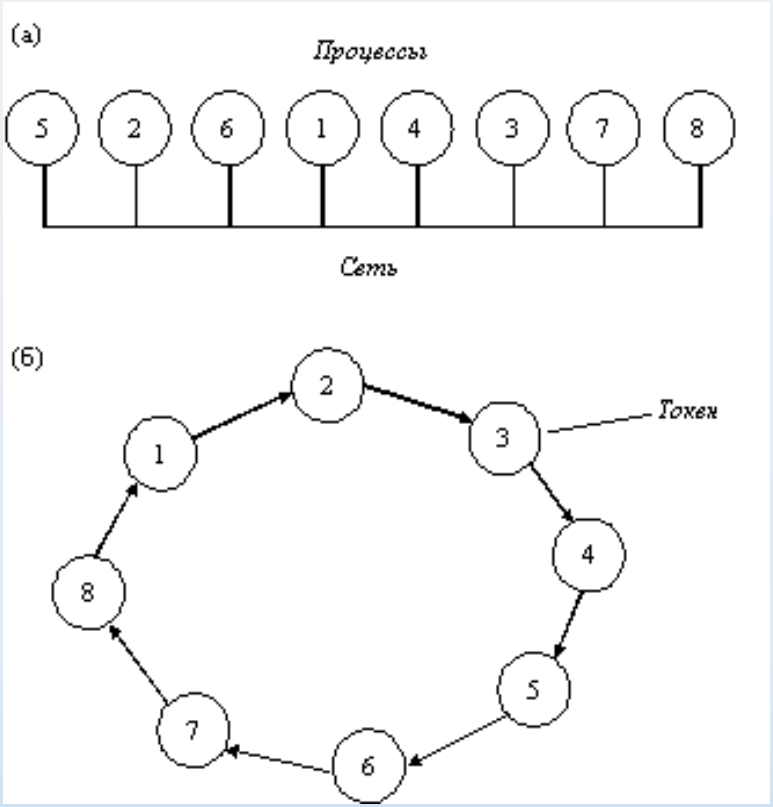


Рисунок 10 – Средства взаимного исключения в распределенных системах:
а – неупорядоченная группа процессов в сети;
б – логическое кольцо, образованное программным обеспечением

Когда кольцо инициализируется, процессу 0 передается маркер (токен). Маркер циркулирует по кольцу. Он переходит от процесса n к процессу n + 1 путем передачи сообщения по типу «точка-точка».

Когда процесс получает токен от своего соседа, он анализирует, не требуется ли ему самому войти в критическую секцию. Если да, то процесс входит в критическую секцию.

После того, как процесс выйдет из критической секции, он передает токен дальше по кольцу. Если же процесс, принявший токен от своего соседа, не заинтересован во вхождении в критическую секцию, то он сразу отправляет токен в кольцо. Следовательно, если ни один из процессов не желает входить в критическую секцию, то в этом случае токен просто циркулирует по кольцу с высокой скоростью.

В алгоритме Token Ring число сообщений переменно: от 1 в случае, если каждый процесс входил в критическую секцию, до бесконечно большого числа, при циркуляции маркера по кольцу, в котором ни один процесс не входил в критическую секцию.

Недостатки:

- если маркер потеряется, то его надо регенерировать. Обнаружить потерю тяжело (время прихода неизвестно).

- если какой-то процесс перестанет функционировать, то алгоритм не работает. Однако восстановление проще, чем в других случаях. Наличие квитанций позволит обнаружить такой процесс в момент передачи маркера (если поломка произошла вне критического интервала). Переставший функционировать процесс должен быть исключен из логического кольца, для этого придется каждому знать текущую конфигурацию кольца.

Выводы

В ходе лекции рассмотрены следующие вопросы:

- общие положения;

- централизованный алгоритм;

- алгоритмы на основе получения разрешений;

- алгоритмы на основе передачи маркера.

Задание на самостоятельную работу

1. Конспект лекций.

Вид и тема следующего занятия

Практическое занятие №7. Dependency Injection (ч. 1)