

3.15 Планирование ввода-вывода. Алгоритмы (лифтовые, с очередями и задержками, предсказывающие, абсолютно честное планирование). Особенности для различных типов устройств (лента, HDD, SSD, виртуальные устройства в памяти). Синдром обмена приоритетами при вводе-выводе, методы его избежания. Latency и производительность

Алгоритмы (на примерах реализованных в ядре Linux планировщиков)

Без планировщика, каждый раз когда происходит запрос на ввод-вывод, происходило бы взаимодействие с ядром и такие операции бы выполнялись немедленно. Более того, может возникнуть такая ситуация, когда вы можете получить огромное количество запросов на ввод-вывод, которое заставит головки диска буквально метаться по нему стороны в сторону. Разница между производительностью жестких дисков и операционной системой выросла очень быстро. Для обслуживания прерывания — приостанавливается работа всех остальных приложений и со стороны это выглядит как снижение отзывчивости системы. Планирование событий ввода-вывода несет в себе необходимость решения многих вопросов. Планировщику необходимо хранить поступившие запросы в специальной очереди.

Самое главное с чего следует начать при рассмотрении архитектуры планировщика или настройки существующих планировщиков — это определение назначения, функций и роли системы.

Есть несколько типичных методов, которые используются планировщиками ввода-вывода:

- Слияние запросов: в рамках этой техники, запросы на чтение-запись схожего назначения объединяются для сокращения количества дисковых операций и увеличение длительности системных вызовов ввода-вывода (что, как правило, приводит к повышению производительности самого ввода-вывода).
- «Алгоритм лифта»: Запросы ввода-вывода упорядочиваются исходя из физического размещения данных на диске таким образом, чтобы головки диска как можно больше перемещались в одном направлении и как можно более упорядоченно.
- Переупорядочение запросов: Эта техника переупорядочивает запросы на основе их приоритета. Алгоритм реализации данной техники зависит от конкретного планировщика.
- Кроме того, почти все планировщики ввода-вывода учитывают фактор «ресурсного голодания», поэтому в итоге (рано или поздно) будут обслужены все запросы (*имеется ввиду, что планировщик следит за тем, чтобы запрос на ввод-вывод слишком долго не находился в очереди, т.е. не «голодал» бы — прим.перев.*).

Сами алгоритмы см. в Приложении.

Особенности для различных типов устройств (лента, HDD, SSD, виртуальные устройства в памяти).

Для **HDD** стараются переупорядочить запросы так, чтобы движение головки осуществлялось в одном направлении.

Для **SSD** дисков (на википедии есть нормальная статья про SSD) чаще всего используют NOOP-планировщик, использующий помимо FIFO очереди только технику слияния, поскольку там нет ни дисков, ни головок по ним бегающих. (Я не очень понимаю, почему бы не использовать технику переупорядочивания запросов на основе приоритетов, но об этом можно умолчать)

Для **ленточных накопителей**

(<http://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D0%B8%D0%BC%D0%B5%D1%80>) (в противоположность SSD) основное время уходит на ожидание прокрутки ленты до места, с которого нужно читать или на которое нужно записывать. Таким образом, насколько я понимаю, очень важно использовать слияние запросов с достаточно большим временем ожидания (чтобы собрать много запросов из близких мест и выполнить их одновременно), что вводит большую Latency (задержку), поэтому используют для их, в основном, для архивации данных, где Latency роли не играет особой.

Для **виртуальных устройств в памяти** (RAM, насколько я понимаю) логично было бы упорядочивать по приоритету процессов + обеспечить отсутствие голодания. Остальные оптимизации, казалось бы, лишены смысла.

Priority inversion and possible solutions

Хорошо (даже с картинками) написано вот здесь: <http://www.netrino.com/Embedded-Systems/How-To/RTOS-Priority-Inversion>, но на англ. википедии тоже нормально.

Собственно, суть простая. Есть два процесса H и L с высоким и, соответственно, низким приоритетом. L начинает чуть раньше и захватывает io-ресурс, который не может освободить сразу же по появлении H, пытается доработать, но тут появляется еще один процесс M ($H > M > L$), которому этот ресурс нафиг не нужен, и спокойно вытесняет L. И вот получилось, что независимые друг от друга H и M, а выполняется процесс с меньшим приоритетом. И процесс может повториться. Так что проблема серьезная. (Говорят, есть еще вероятность dead-locka, но, может быть, это с несколькими ресурсами? Над этим стоит подумать)

Как ее решать?

1. Запретить любые прерывания во время пользования ресурсом (значит M не сможет вытеснить L). Такой вариант плохо подходит для персональных компьютеров, но где-то используется (подробно написано в википедии)
2. Priority ceiling. Каждому ресурсу ставится в соответствие некоторый «потолочный» приоритет – максимальный приоритет, который может иметь процесс, пытающийся захватить ресурс. В момент захвата ресурса процессу присваивается потолочный приоритет этого ресурса, гарантируя, что процесс никто не вытеснит. (есть еще разновидность OCPP, которая как-то поумнее делает, если у тебя осталось время, можно почитать)

3. Наследование приоритетов. Как только появляется процесс А, претендующий на ресурс, текущему владельцу этого ресурса В временно ставится приоритет $\max(\text{Pr}(A), \text{Pr}(B))$, где $\text{Pr}(B)$ – текущий приоритет процесса В (возможно точно так же унаследованный от некоторого ожидающего ресурс процесса С)

Latency и производительность

Если говорить о Latency, то предпочтительным планировщиком является Deadline *или с дополнительными эвристиками Anticipatory* (см. Приложение). Соответственно, на его тему и нужно болтать.

За производительность отвечают все методы, перечисленные в начале. Вряд ли что-то еще нужно рассказывать.

Приложение

В настоящее время в ядре Linux существует четыре(?) планировщика ввода-вывода:

- NOOP *elevator=noop*
- Anticipatory (Упреждающий конвейер) *elevator=as*
- Deadline *elevator=deadline*
- CFQ (Completely Fair Queuing — алгоритм полностью честной очереди) *elevator=cfq*
- BFQ (Budget Fair Queueing I/O Scheduler)

Планировщик NOOP

Планировщик NOOP (no-operate) является самым простым. Он помещает все входящие запросы ввода-вывода в простой буфер типа FIFO (First-In, First-Out — первый вошел, первый вышел) и затем просто запускает их на исполнение. Заметим, что это происходит для всех процессов, существующих в системе, независимо от типа запросов ввода-вывода (чтение, запись, поиск необходимой дорожки т.д.). Также он выполняет нечто подобное слиянию запросов (см. выше).

Согласно этой статье, планировщик NOOP «... использует минимальное количество инструкций процессора на операцию ввода-вывода для выполнения простейшей функциональности: слияние и сортировка запросов». Данный планировщик предполагает, что сами устройства хранения данных будут оптимизировать быстродействие операций ввода-вывода (например, внешний RAID-контроллер или SAN).

Потенциально, планировщик NOOP будет хорошо работать с устройствами хранения данных, которые не имеют механических частей для чтения данных (т.е. головок диска). Причина кроется в том, что данный планировщик не делает никаких попыток оптимизировать движение головок, выполняя простейшее слияние запросов, что также помогает повышению пропускной способности диска. Поэтому такие устройства хранения как флэш-диски, SSD-диски, USB-накопители и т.п., которые имеют очень малое время поиска данных (seek time) могут получить преимущество, используя планировщик NOOP.

Anticipatory IO Scheduler (Предполагающий планировщик)

Anticipatory IO Scheduler, как следует из названия, пытается предположить к каким блокам диска будут выполнены следующие запросы. Он выполняет слияние запросов, простейший однонаправленный алгоритм лифта, объединение (request batching) запросов на чтение и запись. После того как планировщик обслужил запрос на чтение-запись, он предполагает, что следующий запрос будет для последующего блока, приостанавливаясь на небольшое количество времени. Если такой запрос поступает, то он обслуживается очень быстро, поскольку головки диска находятся в нужном месте. Такой подход приносит незначительное замедление в работу системы ввода-вывода по причине необходимости ожидания следующего запроса. Однако этот недостаток может быть компенсирован увеличением производительности запросов к соседним областям диска. Некоторые исследования показали, что anticipatory scheduler работает действительно хорошо при некоторых нагрузках системы. Например, наблюдалось, что веб-сервер Apache может достигать увеличения своей пропускной способности до 71% с помощью данного планировщика. С другой стороны, существуют наблюдения, что данный планировщик вызывал замедление работы баз данных на 15%.

Anticipatory имеет также следующие параметры:

- read_expire – наиболее важный параметр для планировщика deadline, он определяет максимальное время, которое запрос на чтение может ожидать, прежде чем будет обслужен.
- write_expire – максимальное время, которое запрос на запись может ожидать перед тем, как он будет обслужен.
- fifo_batch – число запросов в каждом пакете, который будет послан на немедленное обслуживание в случае, если время обслуживания истекло.
- writes_starved – с помощью этого параметра меняется приоритет, который запрос на операцию чтения имеет над запросом на операцию записи. Поскольку необслуженная операция чтения влияет на характеристики сильнее, чем необслуженная операция записи, обычно операции чтения имеют приоритет над операциями записи.
- front_merges – с помощью этого параметра изменяется значение, установленное по умолчанию, и равное 1, указывающее сколько запросов в начале очереди могут быть объединены вместе, в отличие от запросов, находящихся в конце очереди.
- read_batch_expire – время, потраченное на обслуживание запроса на чтение, перед обслуживанием отложенного запроса на запись.
- write_batch_expire – обратное предыдущему.
- antic_expire – время в миллисекундах, в течение которого планировщик будет находиться в паузе, пока он ожидает следующего запроса от приложения перед тем, как перейдет к обслуживанию следующего запроса.

Deadline IO Scheduler (планировщик предельных сроков)

Deadline IO Scheduler был написан Дженсом Аксбо (Jens Axboe), широко известным разработчиком ядра. Основопологающим принципом его работы является гарантированное время запуска запросов ввода-вывода на обслуживание. Он сочетает в себе такие возможности как слияние запросов, однонаправленный алгоритм лифта и устанавливает предельный срок на обслуживание всех запросов (отсюда и такое название). Он поддерживает две специальные «очереди сроков выполнения» (deadline queues) в дополнение к двум отдельным «отсортированным очередям» на чтение и запись

(sorted queues). Задания в очереди сроков выполнения сортируются по времени исполнения запросов по принципу «меньшее время — более раннее обслуживание — ближе к началу очереди». Очереди на чтение и запись сортируются на основе запрашиваемого ими номера сектора (алгоритм лифта).

Данный планировщик действительно помогает пропускной способности в случаях чтения с секторов с большим номером блока (*на внешних областях диска — прим.перев.*).

Операции чтения могут иногда заблокировать приложения, поскольку пока они выполняются, приложения ждут их завершения. С другой стороны, операции записи могут выполняться гораздо быстрее, поскольку они производятся в дисковый кэш (если только вы не отключили его использование). Более медленные операции чтения с внешних областей диска перемещаются к концу очереди, а более быстрые операции чтения с более близких областей поступят на обслуживание раньше. Такой алгоритм планировщика позволяет быстрее обслужить все запросы на чтение-запись, даже если существуют запросы на чтение с дальних областей диска.

Схема работы планировщика достаточно прямолинейна. Планировщик сначала решает какую очередь использовать первой. Более высокий приоритет установлен для операций чтения, поскольку, как уже упоминалось, приложения обычно блокируются при запросах на чтение. Затем он проверяет истекло ли время исполнения первого запроса. Если так — данный запрос сразу же обслуживается. В противном случае, планировщик обслуживает пакет запросов из отсортированной очереди. В обоих случаях, планировщик также обслуживает пакет запросов следующий за выбранным в отсортированной очереди.

Deadline scheduler очень полезен для некоторых приложений. В частности, в системах реального времени используется данный планировщик, поскольку в большинстве случаев, он сохраняет низкое время отклика (все запросы обслуживаются в короткий временной период). Также было отмечено, что он также хорошо подходит для систем баз данных, которые имеют диски без поддержки TCQ.

В этом планировщике имеются следующие пять настраиваемых параметров:

- `read_expire` – наиболее важный параметр для планировщика `deadline`, он определяет максимальное время, которое запрос на чтение может ожидать, прежде чем будет обслужен.
- `write_expire` – максимальное время, которое запрос на запись может ожидать перед тем, как он будет обслужен.
- `fifo_batch` – число запросов в каждом пакете, который будет послан на немедленное обслуживание в случае, если время обслуживания истекло.
- `writes_starved` – с помощью этого параметра меняется приоритет, который запрос на операцию чтения имеет над запросом на операцию записи. Поскольку необслуженная операция чтения влияет на характеристики сильнее, чем необслуженная операция записи, обычно операции чтения имеют приоритет над операциями записи.
- `front_merges` – с помощью этого параметра изменяется значение, установленное по умолчанию, и равное 1, указывающее сколько запросов в начале очереди могут быть объединены вместе, в отличие от запросов, находящихся в конце очереди.

CFQ IO Scheduler (планировщик с полностью честной очередью)

CFQ (Completely Fair Queue — планировщик с полностью честной очередью) IO scheduler в настоящее время является планировщиком по умолчанию в ядре Linux. Он использует и слияние запросов и алгоритм лифта. При работе CFQ синхронно размещает запросы на

ввод-вывод от процессов в ряд очередей на каждый процесс (per-process queues). Затем он выделяет отрезки времени (timeslices) для каждой очереди на доступ к диску. Длина отрезков времени и количество запросов в очереди, которые будут обслужены, зависит от приоритета ввода-вывода конкретного процесса. Асинхронные запросы от всех процессов объединяются в несколько очередей по одной на каждый приоритет.

Дженс Аксбо (Jens Axboe) является автором планировщика CFQ и включает то, что Дженс называет «алгоритмом лифта Линуса». Добавление данной возможности произошло при необходимости внесения функций для предотвращения «голодания процессов» в наиболее неблагоприятной ситуации, которая может произойти при чтении со внешних дорожек диска. Также приведенная ссылка указывает на хорошую дискуссию по разработке планировщика CFQ и сложностях его дизайна (также в нем обсуждается дизайн deadline scheduler — очень рекомендую это прочесть).

CFQ предоставляет пользователям (процессам) конкретного устройства хранения данных необходимое количество запросов ввода-вывода для определённого промежутка времени. Это сделано для многопользовательских систем, так как все пользователи в таких системах получают один и тот же уровень отклика. Более того, CFQ достигает некоторых из характеристик anticipatory scheduler по хорошей пропускной способности, поскольку позволяет обслуживаемой очереди простаивать какое-то время по завершению выполнения запроса на ввод-вывод в ожидании запроса, который может оказаться близким к только что выполненному запросу (*и к тому же позволяет настройку своих параметров — прим. перев.*).

В этом планировщике имеются следующие параметры:

- quantum — количество внутренних очередей, запросы из которых берутся за один цикл и перемещаются в очередь диспетчера для обработки. Планировщик cfq может иметь 64 внутренние очереди, но он перемещает запросы в очередь диспетчера только при посещении первых восьми внутренних очередей, а затем — следующие восемь в следующем цикле, и т.д..
- queued — максимальное число запросов, допустимое в данной внутренней очереди.