ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В наши дни на фоне высокого спроса на высокоскоростную и надёжную передачу данных остро встаёт проблема обработки трафика и обеспечения должного уровня обслуживания в узлах компьютерных сетей. Крупные производители сетевого оборудования предоставляют эффективные решения обозначенных проблем, однако оборудование и программное обеспечение стоят немалых денег, что делает их недоступными для массового пользователя. Большое распространение в качестве сетевого ПО получили дистрибутивы Linux, которые из-за многолетнего использования имеют богатый набор возможностей в этой области. Таким образом, встаёт вопрос о интеграции возможностей, предоставляемых гигантами индустрии, с широко распространённым открытым вариантом.

В связи с этим было решено изучить дисциплину обслуживания, разработанную компанией Cisco, известным вендором на рынке сетевых решений, и внедрить её в ядро последней стабильной версии Linux. В качестве дисциплины обслуживания был выбран взвешенный алгоритм честного обслуживания основанного на классах (Class-Based Weighted Fair Queuing, CBWFQ).

CBWFQ является расширением функциональности широко известного взвешенного алгоритма честного обслуживания очереди (Weighted Fair Queuing). Он поддерживает определение пользовательских классов трафика на основе ряда критериев соответствия (протокол, входящий интерфейс и т.д.) и назначение их характеристик, которые отвечают за выделяемые классу ресурсы (вес, пропускная способность, максимальное количество пакетов в очереди, задержка). Такой подход предоставляет гибкую настройку распределения пропускной способности канала между классами трафика и оказывается весьма эффективным в передаче данных в сравнении с рядом других популярных дисциплин.[1]

Цель работы – реализовать алгоритм Class-Based WFQ в виде модуля ядра Linux, основываясь на имеющихся описаниях в соответствующей литературе. Для выполнения цели работы необходимо выполнить следующие задачи:

1. Проанализировать и сравнить дисциплины обслуживания PQ, CBQ, HTB, HFSC, FWFQ, CBWFQ.
2. Восстановить алгоритмы Class-Based WFQ.
3. Настроить среду для реализации и тестирования.

4. Реализовать модуль ядра CBWFQ в ядре Linux.
5. Реализовать интерфейс утилиты tc для управления модулем.
6. Провести тестирование.

Сравнительный анализ дисциплин обслуживания в работе основываются на научных статьях, документации и исходных кодах; архитектура подсистемы Linux по управлению качеством обслуживания затрагивается в книге [2], однако в силу отсутствия полной документации все выводы о структуре делались на основе исходного кода ядра Linux. Алгоритм CBWFQ воссоздавался на основе документации Cisco, книг [2] и [3] и исходного кода существующих дисциплин обслуживания.

1 АНАЛИЗ ДИСЦИПЛИН ОБСЛУЖИВАНИЯ

1.1 Дисциплины обслуживания

В теории массового обслуживания дисциплиной обслуживания очередей (ДО) называют правило выбора заявок из очереди для обслуживания[4], определяющее способ отсылки данных. На практике дисциплина обслуживания определяется не только порядком обслуживания, но и способом организации очередей. Очередь – это базовый элемент управления трафиком, являющийся неотъемлемым элементом системы планирования. Обычно, под очередью подразумеют буфер, где пакеты ожидают передачи устройством. [5]

Дисциплины обслуживания делятся на классовые и бесклассовые дисциплины.

1. Бесклассовые дисциплины обслуживания могут принимать трафик, перепланировать его, задерживать или отбрасывать. Этот тип дисциплин обычно используется по умолчанию или для ограничения трафика через узел. Такой тип дисциплин обслуживания малофункционален и имеет множество недостатков; обычно их используют для обслуживания классов в классовых дисциплинах.

2. Классовые дисциплины обслуживания разделяют трафик на классы с помощью процесса классификации, основанного в основном на фильтрах; такой подход позволяет дифференцировать трафик, отдавая канал более приоритезированному.[6]

Рассмотрим и проанализируем наиболее известные дисциплины обслуживания (PQ, CBQ, HTB, HFSC) и дисциплины семейства WFQ (FWFQ и CBWFQ).

1.2 Приоритетные очереди

Приоритетные очереди (Priority Queueing, PQ) – это техника обслуживания, при которой используется множество очередей с разными приоритетами. Очереди обслуживаются в циклическом порядке (алгоритмом Round-robin) от самого высокого до самого низкого приоритета; обслуживание следующей по порядку очереди происходит, если более приоритетные очереди пусты. Каждая очередь внутри обслуживается в порядке FIFO (First-In, First-Out). В случае пе-

реполнения отбрасываются пакеты из очереди с более низким приоритетом.[7] Схема дисциплины представлена на рисунке 1.

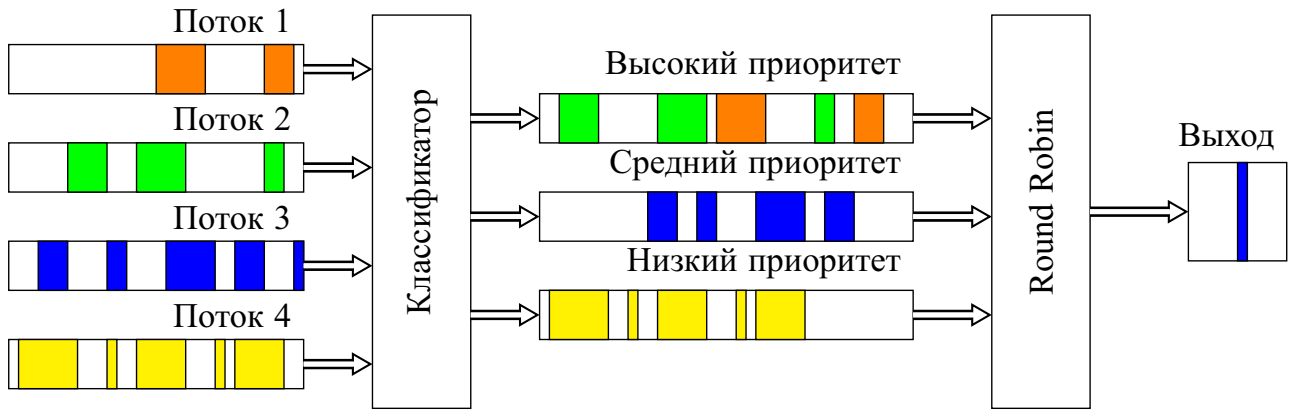


Рисунок 1 — Схема обслуживания алгоритмом приоритетных очередей.

Дисциплина используется, чтобы снизить время отклика, когда нет нужды замедлять трафик[8].

В Linux алгоритм реализован в виде дисциплины prio, которая создаёт фиксированное значение очередей обслуживания (групп или bands), планируемые дисциплиной pfifo_fast, и управляет очередями в соответствии с картой приоритетов.[8]

Преимущества алгоритма состоят в следующем:

- возможность понижения времени отклика, когда нет необходимости замедлять трафик [8];
- наиболее простая в реализации классовая дисциплина обслуживания;
- для software-based маршрутизаторов PQ предоставляет относительно небольшую вычислительную нагрузку на систему в сравнении с более сложными ДО;
- PQ позволяет маршрутизаторам организовывать буферизацию пакетов и обслуживать один класс трафика отдельно от других. [9]

Однако приоритетные очереди обладают рядом существенных недостатков:

- возникает проблема простоя канала (отсутствие обслуживания в течение продолжительного времени) для низкоприоритетного трафика при избытке высокоприоритетного[7];

– избыточный высокоприоритетный трафик может значительно увеличивать задержку и джиттер для менее приоритетного трафика;
– не решается проблема с TCP и UDP, когда TCP-трафику назначается высокий приоритет и он пытается поглотить всю пропускную способность. [9]

1.3 Алгоритм управления очередями на основе классов

Алгоритм управления очередями на основе классов (Class Based Queueing, CBQ) – это классовая дисциплина обслуживания, которая реализует иерархическое разделение канала между классами и позволяет шейпинг трафика.[10]

Главная цель CBQ – это планировка пакетов в очередях, гарантия определённой скорости передачи и разделение канала. Если в очереди нет пакетов, её пропускная способность становится доступной для других очередей. Сила этого метода состоит в том, что он позволяет справляться со значительно различными требованиями к пропускной способности канала среди потоков. Это реализовано путём назначения определённого процента доступной ширины канала каждой очереди. CBQ избегает проблему простоя канала, которой страдает алгоритм PQ, так как как минимум один пакет обслуживается от каждой очереди в течение цикла обслуживания.[7]

Алгоритм CBQ представляет канал в виде иерархической структуры [11] пример которой представлен на рисунке 2. Голубым цветом обозначен узел, представляющий собой основной канал; он разделяется между двумя классами трафика: интерактивным (левый узел) и остальным (правый узел), – которым назначается процент пропускной способности от остального канала. Весь трафик, относящийся к классу, будет получать выделенную пропускную способность для этого класса. Эти классы трафика могут разделяться на подклассы и так далее. Если класс не использует пропускную способность, она будет выделяться классу-соседу. Этот механизм называется механизмом разделения канала [11].

Алгоритм CBQ состоит в следующем. Сначала пакеты классифицируются в классы обслуживания в соответствии с определёнными критериями и сохраняются в соответствующей очереди. Очереди обслуживаются циклически. Различное количество пропускной способности может быть назначено для каждой очереди двумя различными способами: с помощью позволения очереди отправлять

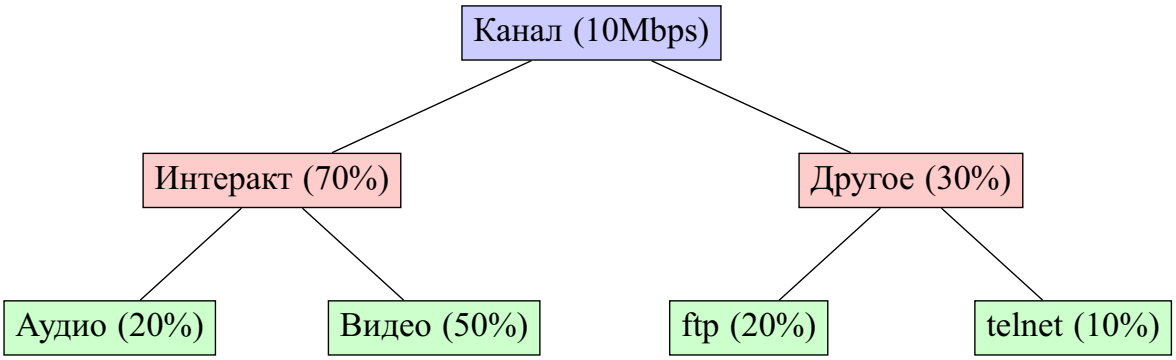


Рисунок 2 — Пример работы механизма разделения канала.

более чем один пакет на каждый цикл обслуживания или с помощью позволения очереди отправлять только один пакет за цикл, но при этом очередь может быть обслужена несколько раз за цикл.[7]

Вычисления в CBQ основываются на вычислении времени в микросекундах между запросами, на основе которого рассчитывается средняя загруженность канала; в этом и состоит главная проблема неточности CBQ в Linux.[6]

Преимущества алгоритма состоят в следующем:

– позволяет контролировать количество пропускной способности для каждого класса обслуживания;
– каждый класс получает обслуживание, вне зависимости от других классов. Это помогает избегать проблемы PQ, когда при избытке высокоприоритезированного трафика низкоприоритезированный не обслуживался вообще.[7]

Недостатки же в большей следуют из особенностей реализации алгоритма в системе Linux:

– честное выделение пропускной способности происходит, только если пакеты из всех очередей имеют сравнительно одинаковый размер. Если один класс обслуживания содержит пакет, который длиннее остальных, этот класс обслуживания получит большую пропускную способность, чем сконфигурированное значение [7];
– высокая сложность реализации. В ядре Linux реализация CBQ приближённая и в некоторых случаях может давать неверные результаты.[6]

1.4 Алгоритм иерархического маркерного ведра

Алгоритм иерархического маркерного ведра (Hierarchical Token Bucket, HTB) – дисциплина обслуживания с иерархическим разделением канала между классами.

HTB, подобно CBQ, использует механизм разделения канала. HTB обеспечивает, что количество обслуживания, предоставляемое каждому классу, является, минимальным значением из запрошенного количества и назначенного классу. Когда класс запрашивает меньше, чем ему выделено, оставшаяся пропускная способность распределяется между другими классами, которые требуют обслуживание.[12]

Отличительная особенность HTB от CBQ состоит в том, что в HTB принцип работы основывается на определении объема трафика[6], что даёт более точные результаты.

HTB состоит из произвольного числа иерархически организованных фильтров маркерного ведра (Token Bucket Filter, TBF)[7], однако реализация не использует готовый модуль tbf: алгоритм маркерного ядра встроен в код реализации HTB, что повышает его эффективность. Внутренние классы содержат фильтры, которые распределяют пакеты по очередям и метаинформацию для разделения канала. Листовые классы содержат очереди, которые содержат очереди, которые управляются сконфигурированными дисциплинами обслуживания (по умолчанию pfifo_fast). Пример сконфигурированного дерева HTB представлен на рисунке 3.

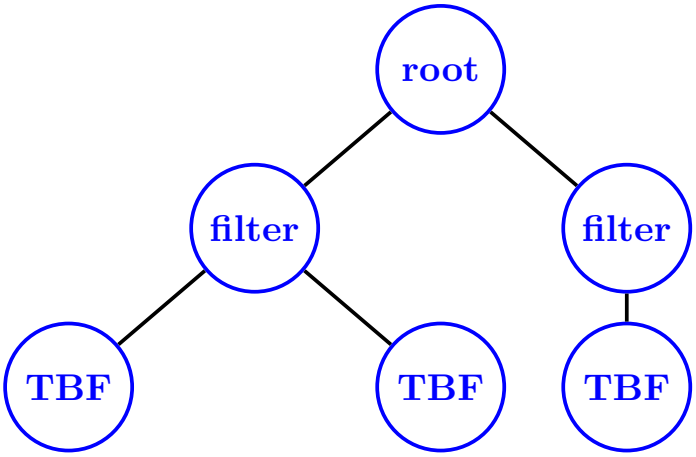


Рисунок 3 — Пример иерархии классов при использовании дисциплины HTB.

При добавлении пакета в очередь HTB начинает обход дерева от корня для определения подходящей очереди: в каждом узле происходит поиск инструкций, и затем происходит переход в узел, на который ссылается инструкция. Обход заканчивается, когда алгоритм доходит до листа, в очередь которого помещается пакет.[13] В реализации алгоритма существует прямая очередь, которая используется не только в качестве очереди с наивысшем приоритетом, но и как очередь, в которую попадают пакеты, не определённые в другую очередь. Это мера не самая удачная, но используется для избежания ошибок.

Преимущества алгоритма HTB приведены ниже.

- Наиболее используемая дисциплина обслуживания в Linux, так как HTB эффективно справляется с обработкой пакетов, а конфигурация HTB легко масштабируется.[6]
- Иерархическая структура предоставляет гибкую возможность конфигурировать трафик.
- Не зависит от характеристик интерфейса и не нуждается в знании о лежащей в основе пропускной способности выходного интерфейса из-за свойств TBF. [13]
- Вычислительно проще, чем алгоритм CBQ.[12]

Недостатки.

- Медленнее CBQ в N раз, где N – глубина дерева разделения, что, однако, компенсируется простотой вычислений.[12]

1.5 Алгоритм иерархических честных кривых обслуживания

HFSC (Hierarchical Fair-Service Curve) – иерархический алгоритм планирования пакетов, основанный на математической модели честных кривых обслуживания (Fair Service Curve), где под термином "кривая обслуживания"подразумевается зависящая от времени неубывающая функция, которая служит нижней границей количества обслуживания, предоставляемого системой.[14]

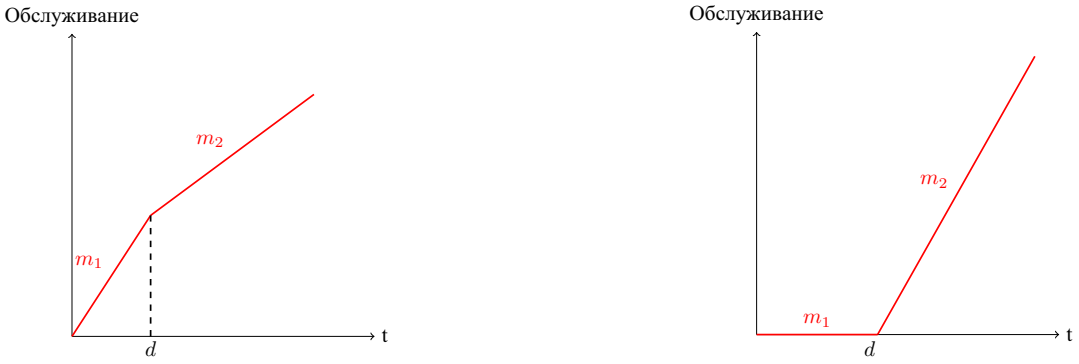
HFSC ставит перед собой цели:

- гарантировать точное выделение пропускной способности и задержки для всех листовых классов (критерий реального времени);

– честно выделять избыточную пропускную способность так, как указано классовой иерархией (критерий разделения канала);

– минимизировать несоответствие кривой обслуживания идеальной модели и действительного количество обслуживания.[15]

Алгоритм планировки основан на двух критериях: критерий реального времени (real-time) и критерий разделения канала (link-sharing). Критерии реального времени используются для выбора пакета в условиях, когда есть потенциальная опасность, что гарантия обслуживания для листового класса нарушается. В ином случае используется критерий разделения канала.[15]



(a) Вогнутая кривая.

(b) Выпуклая кривая.

Рисунок 4 — Примеры кривых обслуживания. m_1 – скорость в стационарном состоянии, m_2 – скорость в режиме burst, d – время, за которое происходит передача в режиме burst.[16]

На рисунке 4 изображены примеры кривых обслуживания, используемых в дисциплине HFSC; параметры m_1, m_2 и d, отображённые на графиках, задаются при конфигурации дисциплины.[16]

HFSC использует три типа временных параметров: время крайнего срока (deadline time), "подходящее"время (eligible time) и виртуальное время (virtual time). Время крайнего срока назначается таким образом, чтобы, если крайние сроки всех пакетов сессии выполнены, его кривая была гарантирована. "Подходящее"время используется для выбора критерия планировки для следующего пакета. Виртуальное время показывает нормализованное количество обслуживания, которое получил класс. Виртуальное время присуще всем вершинам дерева классов, так как является важным параметром при критерии разделение канала, при котором должно минимизироваться несоответствие между виртуальным

временем класса и временами его соседей (так как в идеальной модели виртуальное время соседей одинаково); при выборе критерия разделения канала алгоритм рекурсивно, начиная с корня, обходит всё дерево, переходя в вершины с наименьшим виртуальным временем. Время крайнего срока и «подходящее» время используются дополнительно в листовых классах, так как в этих вершинах непосредственно содержатся очереди.[14]

Основное преимущество алгоритма состоит в том, что он основан на формальной модели с доказанными нижними границами. Он даёт гарантированные результаты и вычисляет более точно, чем дисциплины CBQ и HTB, которые служат схожим целям.

Главные же недостатки HFSC заключены в его достоинстве. Алгоритм основан на формальной модели и имеет множество параметров, требующих дополнительных расчётов и времени на подготовку к конфигурации. Также он довольно сложен в реализации и поддержке.

1.6 Взвешенный алгоритм честного обслуживания очередей

WFQ (Weighted Fait Queueing) – динамический метод планировки пакетов, который предоставляет честное разделение пропускной способности всем потокам трафика. WFQ применяет вес, чтобы идентифицировать и классифицировать трафик в поток и определить, как много выделить пропускной способности каждому потоку относительно других потоков. WFQ на месте планирует интерактивный трафик в начало очереди, уменьшая там самым время ответа, и честно делит оставшуюся пропускную способность между остальными потоками. [17]

WFQ представляет собой аппроксимацию обобщённой схемы разделения процессорного времени (General Processor Sharing, GPS). GPS – это схема, обеспечивающая честное обслуживание по типу взвешенной максиминной схемы равномерного распределения ресурсов (схема, при которой каждому пользователю назначается определённый вес и выделяется равномерная доля ресурсов, пропорциональная этому весу). В соответствии со схемой GPS каждый поток трафика помещается в собственную логическую очередь, после чего бесконечно малый объём данных из каждой непустой очереди обслуживается по круговому принципу. Необходимость обработки бесконечно малого объёма данных на каждом круге обусловлена требованием обслуживания всех непустых очередей на любом конечном временном интервале. Из-за чего схема GPS является спра-

ведливой в любой момент времени. Однако технически невозможно реализовать данную схему на практике; WFQ же предлагает брать за рабочую единицу пакет.[3]

Алгоритм WFQ использует концепцию виртуального времени, чтобы отслеживать расчёты схемы GPS. В рамках этой концепции используется понятие события j, которое обозначает прибытие или отправление пакета во время t_j. Виртуальное время идёт не с той же скоростью, с какой идёт реальное; его скорость зависит от активных потоков в рассматриваемый реальный промежуток времени. Разница представлена на рисунке 5.

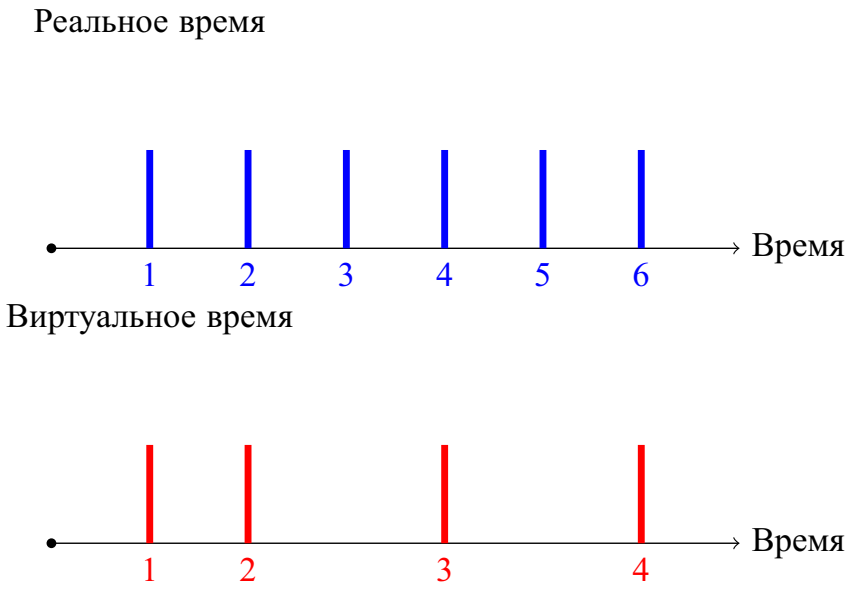


Рисунок 5 — Сравнение движений реального и виртуальных времён.

По Формуле (1) рассчитывается движение виртуального времени в GPS. Скорость виртуального времени зависит от количества активных сессий.

$$V_{t_{j-1}+\tau} = V_{t_{j-1}} + \frac{\tau}{\sum\limits_{i \in B_j} w^i}, \tau \leq t_j - t_{j-1} \tag{1}$$

$$V(0) = 0 \tag{2}$$

где

w^i — вес потока i.

B_j — обозначает, является ли поток на данный момент активным.

Виртуальное время завершения обслуживания обозначает виртуальное время, когда должно завершиться обслуживания в соответствии со схемой GPS. Пакет с наименьшим виртуальным временем первым покинет систему обслуживания. Для вычисления виртуального времени завершения обслуживания по Формуле (4) необходимо вычислить виртуальное время начала обслуживания по Формуле (3).

$$S_i^k = \max\{F_i^{k-1}, V_{a_i^k}\}, \tag{3}$$

где

S_i^k — виртуальное время начала обработки пакета под номером k из потока i.

F_i^k — виртуальное время конца обработки пакета под номером k из потока i.

a_i^k — время прибытия пакета под номером k из потока i.

$$F_i^k = S_i^k + \frac{L_i^k}{w_i}, \tag{4}$$
$$F_i^0 = 0,$$

где

L_i^k — длина пакета под номером k из потока i.

w_i — вес потока i.

В Таблице 1 приведён пример вычисления виртуального времени заверше-

ния обслуживания. По столбцу F видно, что систему раньше покинут пакеты с большим весом.

Таблица 1 — Пример вычисления времени завершения обслуживания.

	Поток 1			Поток 2		
Вес	$w_1 = \frac{1}{3}$			$w_2 = \frac{2}{3}$		
Номер пакета	a_1	L_1	F_1	a_2	L_2	F_2
1	1	1	4	0	3	4
2	2	1	5	3	2	8
3	4	2	9	5	2	14
4	7	2	13	-	-	-

Для отправки каждый цикл обслуживания выбирается пакет с наименьшим виртуальным временем конца обработки. Итоговая пропускная способность r_i рассчитывается на основе заданных весов по Формуле 5[18]

$$r_i = \frac{w_i}{\sum\limits_{i=0}^{N} w_i} \cdot R \tag{5}$$

где

w_i — вес, назначенный потоку i.

N — количество потоков.

R — полная пропускная способность канала.

Рассмотрим алгоритм WFQ на основе потока (Flow-based WFQ, FWFQ). Название алгоритма отсылает к методу классификации пакетов, при котором очередь выделяется для пакетов одного потока.[3] Схема планировщика представлена на рисунке 6.

В реализации FWFQ от Cisco используется алгоритм WFQ на основе порядкового номера пакета. Алгоритм поддерживает два счётчика: счётчик цикла, который определяет количество пройденных циклов побайтового планировщика (и равняется порядковому номеру последнего обслуженного пакета), и значение наибольшего порядкового номера пакета, поставленного в очередь потока. На основе этих значений высчитывается поряд-

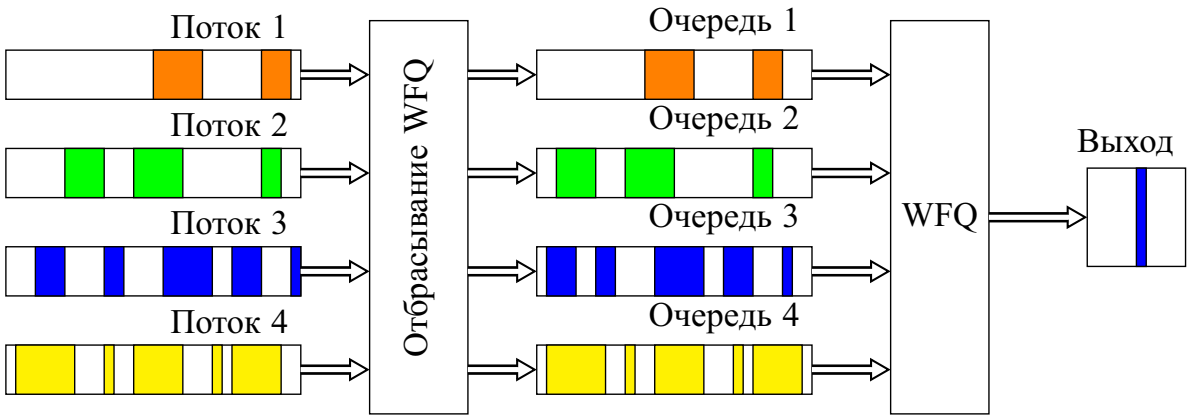


Рисунок 6 — Схема WFQ системы на основе потоков.

ковый номер пакета, пакет с минимальным значением покидает систему. Вычисление порядкового номера пакета вычисляется в зависимости от того, активный ли поток на момент прибытия нового пакета. Если поток был неактивным, то *порядковый номер пакета = размер пакета в байтах · вес + значение счётчика цикла на момент поступления пакета*; если поток активный, то *порядковый номер пакета = размер пакета в байтах · вес + значение наибольшего порядкового номера пакета в потоке*. Вес пакета строко зависит от его приоритета (определяется по соответствующему полю заголовка IP) и не может быть изменён.[3] Условная схема работы алгоритма представлена на рисунке 7.

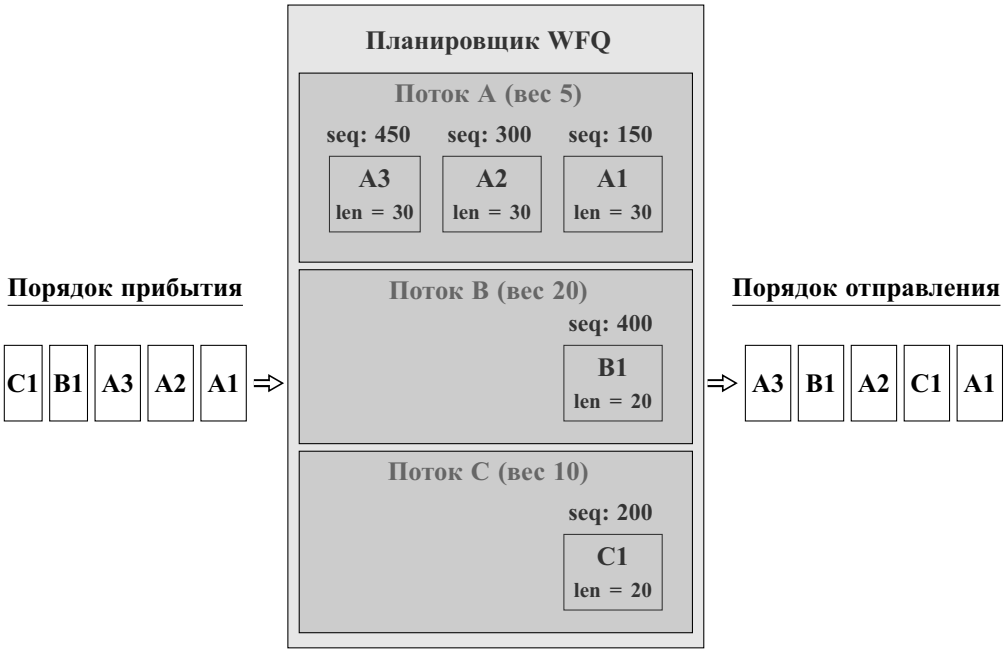


Рисунок 7 — Условная схема работы алгоритма WFQ на вычисления порядкового номера.

Планировщик не нарушает порядка обработки пакетов, принадлежащих одному потока, даже в том случае, если они имеют различный приоритет. В целях планировки в WFQ длина очереди измеряется не в пакетах, а во времени, которое заняла бы передача всех пакетов в очереди.[3]

WFQ использует два метода отбрасывания пакетов: ранее (Early Dropping) и агрессивное (Aggressive Dropping) отбрасывания. Ранее отбрасывание срабатывает тогда, когда достигается congestive discard threshold (CDT); CDT – это количество пакетов, которые могут находиться в системе WFQ перед тем, как начнётся отбрасывание новых пакетов из самой длиной очереди; используется, чтобы начать отбрасывание пакетов из наиболее агрессивного потока, даже перед тем, как он достигнет предел hold queue out (HQO). HQO – это максимальное количество пакетов, которое может быть во всех выходящих очередях в интерфейсе в любое время; при достижении HQO срабатывает агрессивный режим отбрасывания. Алгоритм представлен на рисунке 8. [19]

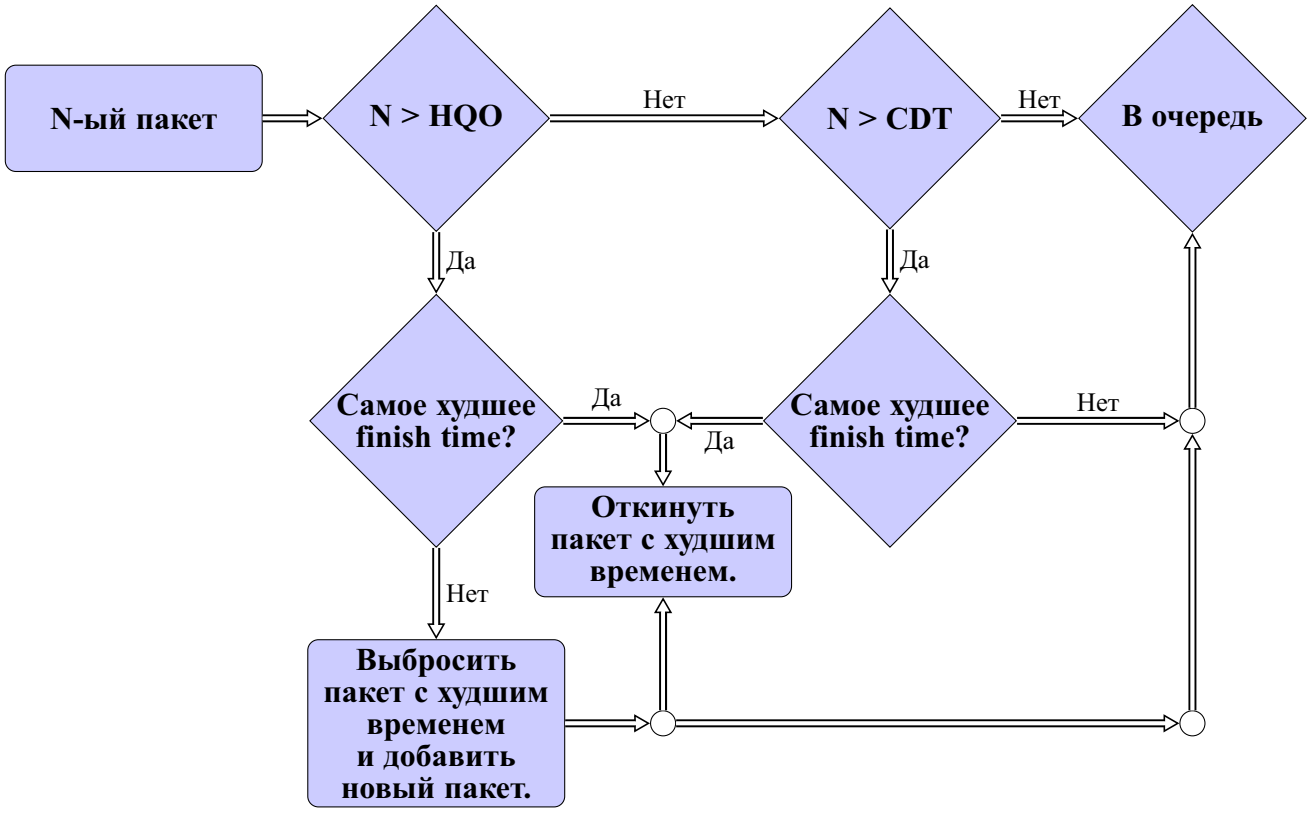


Рисунок 8 — Схема отбрасывания пакетов WFQ.

Преимущества WFQ.

– Простая конфигурация.

– Отбрасывание пакетов из более агрессивных потоков, что предотвращает перегрузки.
– Из-за честного обслуживания отсутствует проблема голодания потоков.

WFQ страдает от нескольких недостатков.

– Трафик не может регулироваться на основе достигнется определённых классов обслуживания.
– Не поддерживает задание определённой пропускной способности для типа трафика.
– В Cisco системах WFQ поддерживается только на медленных каналах.[20]

Эти ограничения были исправлены CBWFQ.

1.7 Взвешенный алгоритм честного обслуживания очередей на основе классов

CBWFQ (Class-based weighted fair queueing) – основанный на классах взвешенный алгоритм равномерного обслуживания очередейс[3]; является расширением функциональности дисциплины обслуживания WFQ, основанной на потоках, для предоставления определяемых пользователями классов трафика.

Class-Based WFQ – это механизм, использующийся для гарантировании пропускной способности для класса. Для CBWFQ класс трафика определяется на основе заданных критериев соответствия: список контроля доступа (ACL), протокол, входящий интерфейс и т.п. Пакеты, удовлетворяющие критериям класса, составляют трафика для этого класса. Дисциплина позволяет задавать до 64-х пользовательских классов.

Схема дисциплины обслуживания представлена на рисунке 9.

После определения класса, ему назначаются характеристики, которые определяют политику очереди: пропускная способность, выделенная классу, максимальная длина очереди и так далее. Алгоритм CBWFQ позволяет явано указать требуемую минимальную полосу пропускания для каждого класса трафика. Полоса пропускания используется в качестве веса класса. Вес можно задать в абсолютной (команда “bandwidth”), в процентной (команда “bandwidth percent”) и в доле от оставшейся полосы пропускания (команда “bandwidth

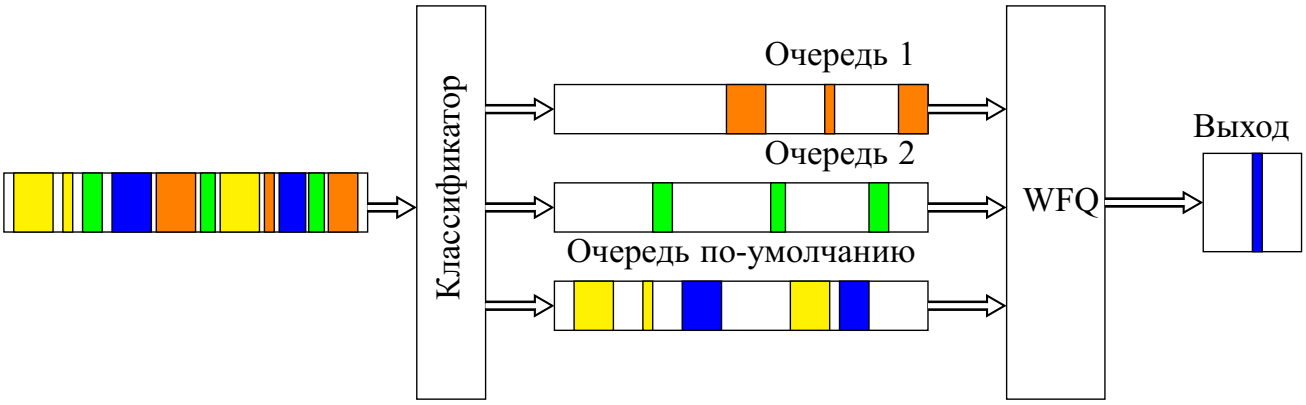


Рисунок 9 — Схема дисциплины обслуживания CBWFQ.

remaining precent”) величинах. Кроме пользовательских классов CBWFQ предоставляет стандартный класс (default class), в который попадает весь трафик, который не был классифицирован. В стандартном классе управление очередью может осуществляться с помощью алгоритмов FIFO и FQ (Fair Queueing). [17]

В случае переполнения очередей начинает работать алгоритм отбрасывания пакетов. В качестве политики отбрасывания пакетов по умолчанию используется отбрасывание конца очереди (Tail Drop), однако допускается сконфигурировать работу алгоритм взвешенного произвольного раннего обнаружения (Weighted Random Early Detection, WRED) для каждого класса.[17]

Преимущества:

- позволяет явно задать полосу пропускания для класса;
- позволяет создавать классы трафика и настраивать их в соответствии с требованиями;
- простая конфигурация вследствие небольшого числа параметров.[3][17]

Недостатки:

- нет поддержки работы с интерактивным трафиком (что исправляется в дисциплине обслуживания Low Latency Queueing (LLQ), которая является развитием CBWFQ);
- в Cisco реализации наблюдается ограничение на количество пользовательских классов (до 64-х классов);[3]
- отсутствие открытой реализации, что усложняет реализацию алгоритма в других системах и требует его полного воссоздания на основе имеющихся источников.

1.8 Выводы

Таблица 2 — Сравнительная таблица дисциплин обслуживания.

Свойство	**PQ**	**CBQ**	**HTB**	**HFSC**	**FWFQ**	**CBWFQ**
Метод планирования	RR	WRR	RR	RT/LS	WFQ	WFQ
Отбрасывание	TD	TD	TD	TD	ED/AD	TD/WRED
Честность (справедливость)	-	-	-	-	+	+
Разделение канала	-	+	+	+	-	-
Решение проблемы голодания	-	+	+	+	+	+
Сложность реализации	Низк	Выс	Сред	Выс	Сред	Сред
Сложность конфигурации	Низк	Выс	Сред	Выс	Низк	Низк
Конфигурация классов	-	+	+	+	-	+
Реализация в Linux	+	+	+	+	-	-

Каждая из рассмотренных ДО обладает своими достоинствами и недостатками. В Таблице 2 приведено сравнение основных элементов проанализированных дисциплин обслуживания.

Особое внимание следует уделить механизмам честного обслуживания и разделения канала. Оба метода служат решением проблемы голодания, однако используют разные по сложности подходы. Механизм разделения канала использует сложные вычисления и требует не только тщательной реализации и поддержки (что явно видно при исследовании исходного кода приведённых выше дисциплин CBQ, HTB и HFSC), но и кропотливой конфигурации. В то время, когда честное обслуживание не требует реализации сложных механизмов. У него есть свои недостатки: оно не позволяет построения сложных иерархических связей и разделения пропускной способности между классами одного уровня. Однако это требуется не во всех ситуация; для простой конфигурации в нетривиальных, но не требующих тщательного контроля, случаях лучше всего

использовать несложные в конфигурации механизмы, дающие схоЖий результат. Поэтому реализация CBWFQ в ядре Linux целесообразна.

2 РЕАЛИЗАЦИЯ CLASS-BASED WFQ В ЯДРЕ LINUX

2.1 Описание устройства подсистемы планировки в ядре Linux

В операционной системе Linux дисциплина обслуживания, обозначаемая термином qdisc, используется для выбора пакетов из выходящей очереди для отправки на выходной интерфейс. Схема движения пакета приведена на Рисунке 10. Выходная очередь обозначена термином egress; именно на этом этапе следования пакета и работает механизм qdisc.[6]

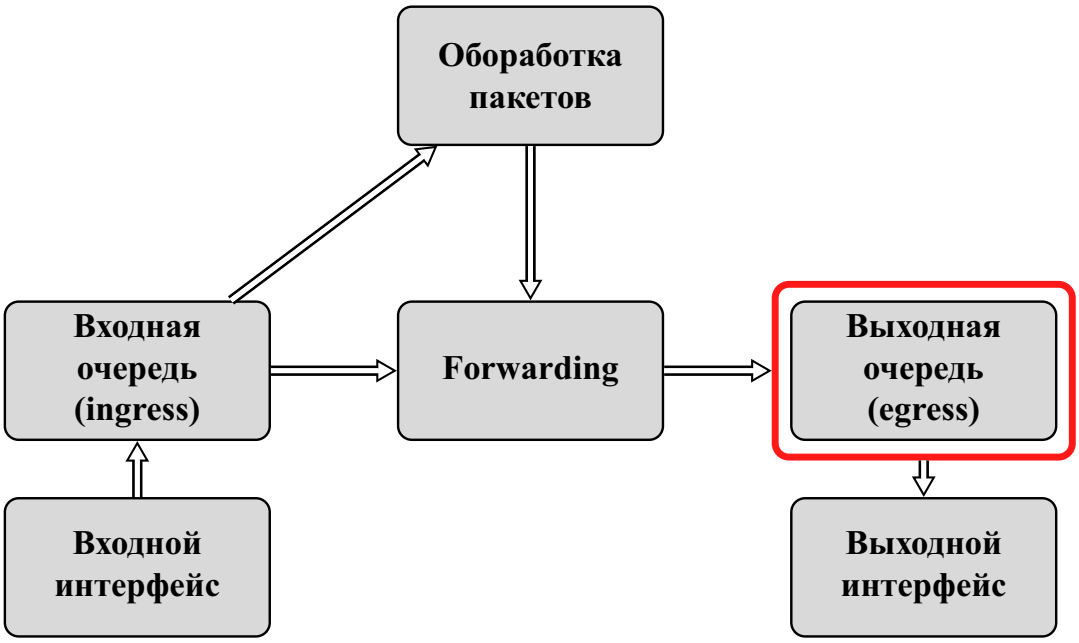


Рисунок 10 — Схема движения пакета в системе Linux[2]. Красным отмечена стадия, в которой работает механизм контроля качества обслуживания.

В общем случае, дисциплина обслуживания — это чёрный ящик, который может принимать поток пакетов и выпускать пакеты, когда устройство готово к отправке, в порядка и во время, определёнными спрятанным в ящике алгоритмом. В ядре Linux дисциплины обслуживания представляются в качестве модулей ядра, которые реализуют предоставляемый ядром интерфейс.

Linux поддерживает классовые и бесклассовые дисциплины обслуживания. Примером бесклассовой дисциплины служит pfifo_fast, классовой — htb.[6]

Классы представляют собой отдельные сущности в иерархии основной дисциплины. Если структура представляет собой дерево, то в классах-узлах мо-

гут содержаться фильтры, которые определят пакет в нужный класс-потомок. В классах-листьях непосредственно располагаются очереди, которые управляются внутренней дисциплиной обслуживания. По умолчанию это pfifo_fast, но можно назначить другие.

Каждый интерфейс имеет корневую дисциплину, которой назначается идентификатор (handle), который используется для обращения к дисциплине. Этот идентификатор состоит из двух частей: мажорной (MAJ) и минорной (MIN); мажорная часть определяет родителя, минорная — непосредственно класс. На Рисунке 11 представлен пример иерархии.

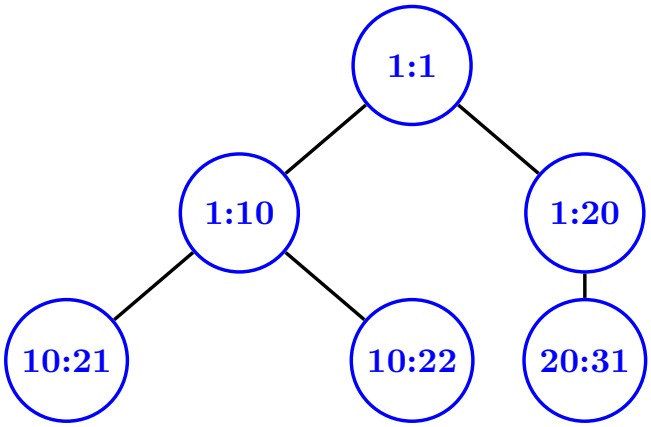


Рисунок 11 — Схема классовой иерархии с использованием идентификаторов MAJ:MIN.

Идентификатор класса называется classid (к примеру, 1:10), а идентификатор его родителя — parenid (1:1 для классов 1:10 и 1:20). По этим идентификаторам происходит поиск нужного класса внутри дисциплины.

Такая иерархия позволяет организовать гибкую систему классификации с набором классов и их подклассов, пакеты в которые назначаются фильтрами, которые предоставляются ядром.

2.2 Интерфейс управления трафиком

В Linux управление трафиком осуществляется с помощью подсистемы Traffic Control, которая предоставляет пользовательский интерфейс с помощью утилиты tc. tc — это пользовательская программа, которая позволяет настраивать дисциплины обслуживания в Linux. Она использует Netlink в качестве коммуникационного канала для взаимодействия между пользовательским пространством

и пространством ядра. tc добавляет новые дисциплины обслуживания, классы трафика, фильтры и предоставляет команды для управление всеми обозначенными объектами.[2]

tc предоставляет интерфейс для дисциплины обслуживания, представленный структурой struct qdisc_util , которая описывает функции для отправления команд и соответствующих параметров ядру и вывода сообщений о настройки дисциплины, списках классов и их настройки, а также статистику от ядра. Сообщение, помимо общей информации для подсистемы, содержит специфичную для дисциплины структуру с опциями, описываемую в заголовке ядра pkt_sched.h:

- структура struct tc_cbwfq_glob определяет глобальные настройки дисциплины обслуживания; структура передаётся по Netlink к модулю дисциплины при инициализации и изменении настроек;
- структура struct tc_cbwfq_copt определяет настройку класса и используется при добавлении или изменения класса.

Патч для заголовка в Приложении А.

Для назначения новой дисциплины обслуживания на интерфейс используется команда “tc qdisc add” системными параметрами (к примеру, название интерфейса), названием дисциплины и её локальными параметрами, которые определяются и обрабатываются в модуле дисциплины для утилиты tc. Для внесения изенений и удаления используются соответственно “tc change” и “tc delete”.

Опции для настройки дисциплины обслуживания:

- “bandwidth”— пропускная способность B канала;
- “default” — ключевое слово, определяющее, что далее пойдёт настройка класса по умолчанию; опции те же самые, что и при настройке класса.

Для классовых дисциплин используется команда “tc class” с под командами “add”, “change” и так далее. Классы обычно имеют параметры, отличные от параметров всей дисциплины обслуживания, поэтому нуждаются в отдельной структуре данных и функции обработчике.

Опции для настройки класса:

- “rate”— минимальная пропускная способность для класса; задаётся в единицах скорости (Mbps, Kbps, bps) или в процентах от размера канала (при использовании ключевого слова “percent”);

– “limit”— максимальное число пакетов в очереди.

Модуль для утилиты tc и ядра Linux, обеспечивающие взаимодействие между пользовательским пространством и дисциплиной представлен в Приложении Б.

2.3 Описание интерфейса

API ядра для подсистемы qdisc предоставляет две функции: register_qdisc (struct Qdisc_ops ∗ops) и обратную — unregister_qdisc (struct Qdisc_ops ∗ops), которые регистрируют и разрегистрируют дисциплину обслуживания на интерфейсе. Важно отметить, что обе эти функции принимают в качестве аргумента структуру struct Qdisc_ops, которая явным образом идентифицирует дисциплину обслуживания в ядре.

Структура struct Qdisc_ops помимо метаинформации (в виде наименования дисциплины) содержит указатели на функции, которые должен реализовывать модуль дисциплины обслуживания для работы в ядре. Если не реализовать некоторые функции, то ядро в некоторых случаях попробует использовать функции по умолчанию, однако для особенно важных (к примеру, изменение конфигурации дисциплины или класса) сообщит пользователю, что операция не реализована.

Поля структуры Qdisc_ops представляют собой указатели на функции представленными ниже сигнатурами.

– enqueue

  int enqueue(struct sk_buff ∗skb, struct Qdisc ∗sch, struct sk_buff ∗∗ to_free);

  Функкция добавляет пакет в очередь. Если пакет был отброшен, функция возвращает код ошибки, говорящий о том, был отброшен пришедший пакет или иной, чьё место занял новый.

– dequeue

  struct sk_buff ∗dequeue(struct Qdisc ∗sch);

  Функция, возвращающая пакет из очереди на отправку. Дисциплина может не передавать пакет при вызове этой функции по решению алгоритма, в таком случае вернув нулевой указатель; однако то же значение алгоритм

возвращает в случае, если очередь пуста, поэтому в таком случае дополнительно проверяется длина очереди.

– peek

struct sk_buff *peek(struct Qdisc *sch);

Функция возвращает пакет из очереди на отправку, не удаляя его из реальной очереди, как это делает функция dequeue.

– init

int init (struct Qdisc *sch, struct nlattr *arg);

Функция инициализирует вновь созданный экземпляр дисциплины обслуживания sch. Вторым аргументом функции является конфигурация дисциплины обслуживания, передаваемая в ядро с помощью подсистемы Netlink.

– change

int change(struct Qdisc *sch, struct nlattr *arg);

Функция изменяет текущие настройки дисциплины обслуживания.

– dump

int dump(struct Qdisc *sch, struct sk_buff *skb);

Функция отправляет по Netlink статистику дисциплины обслуживания.

Также структура содержит указатель на struct Qdisc_class_ops, которая описывает указатели функции исключительно для классовых дисциплин. Ниже приведены наиболее важные сигнатуры и их описания.

– find

unsinged long find(struct Qdisc *sch, u32 classid);

Функция возвращает приведённый к unsinged long адресс класса по его идентификатору (classid).

– change

int change(struct Qdisc *sch, u32 classid , u32 parentid , struct nlattr * attr , unsinged long *arg);

Функция используется для изменения и добавления новых классов в иерархию классов.

– tcf_block, bind_tcf, unbind_tcf

В данном случае, описание сигнатур не даст какой-либо значимой инфор-

мации; практически для всех дисциплин обслуживания они идентичны. Эти функции предназначаются для работы системы фильтрации.

– dump_class

  int dump_class(struct Qdisc ∗sch, unsinged long cl, struct sk_buff ∗skb , struct tcmsg ∗tcm);

  Функция предназначается для передачи по Netlink информации о классе и дополнительной статистики, собранной во время функционирования класса.

Для классовых дисциплин, помимо описанного, реализуют классификацию пакетов, которая определяет класс, куда попадаёт пакет. Классификация обычно выражается в функции classify , которая вызывается при добавлении пакета в очередь (функция enqueue) определяет, какому классу принадлежит пакет, и возвращает указатель на этот класс. Экземпляр структур для дисциплины обслуживания CBWFQ приведён в патче, представленном в Приложении В.

2.4 Алгоритм CBWFQ

Реализация CBWFQ требует:

– вычисление порядкового номера для каждого пакета в очередях и содержание глобального счётчика циклов;

– поддержку классов и классовых операций;

– фильтрацию для классификации трафика по классам.

2.4.1 Структуры хранения данных Class-Based WFQ

Обычно классовые дисциплины обслуживания содержат две основные структуры: для описания непосредственно дисциплины и для описания класса. Структура дисциплины содержит в себе данные, которые описывают всю дисциплину: это могут быть структура данных с классами (в виде списка или дерева), ограничения на очереди, статистика по всей дисциплине и так далее. Структура класса, соответственно, содержит непосредственно очередь и описывающие класс параметры.

Описание полей структуры дисциплины обслуживания struct cbwfq_sched_data.

– struct Qdisc_class_hash clhash
  Хэш-таблица для хранения классов.

– struct tcf_proto ∗ filter_list и sutrct tcf_block ∗block
  Структуры для хранения и обработки фильтров. Используются при выборе класса, в который поместить пакет.

– struct cbwfq_class ∗default_queue
  Ссылка на очередь по умолчанию для быстрого доступа.

– enum cbwfq_rate_type rtype
  Определяет тип, в котором указаны пропускная способность канала и пропускная способность для класса:

  – TCA_CBWFQ_RT_BYTE — ПС задана в байтах;
  – TCA_CBWFQ_RT_PERCENT — ПС задана в процентах.

  Используется для поддержание консистентности конфигурации.

– u32 ifrate
  Пропускная способность канала.

– u32 active_rate
  Суммарная пропускная способность всех классов. Используется для определения состояния системы, при котором ни один поток не активный.

– u64 sch_sn
  Счётчик циклов; используется для определения порядкового номер пакета, принадлежащий неактивному классу.

Описание структуры полей класса.

– struct Qdisc_class_common common
  Структура, использующаяся для управления в хэш-таблице clhash. Содержит в себе идентификатор класса (classid) и метаинформацию для таблицы.

– struct Qdisc ∗queue
  Внутренняя дисциплина обслуживания, непосредственно содержащая пакеты. Настраивается на дисциплину pfast_fifo.

– u32 limit
  Максимальное количество пакетов в очереди класса.

– u32 rate

Минимальна пропускная способность, выделенная классу.

– u64 cl_sn

Значение последнего порядкового номера пакета в очереди класса; используется для определение порядкового номера в случае, когда класс активен.

– bool is_active

Флаг активности класса.

Определение структур и реализация функций представлены в Приложении В.

2.4.2 Добавление пакета в очередь

Алгоритм добавления пакета обычно состоит из схожих действий: классификация и добавление в очередь, если есть место в очереди для пакета. Ниже приведен алгоритм на псевдо-языка.

```
1:  function ENQUEUE(Q, pkt)
2:      c ← CLASSIFY(Q, pkt)
3:      if c.queue_len < c.limit then
4:          DROP(Q, pkt)
5:      else if Q.ENQUEUE(pkt) then
6:          vl ← pkt.len · Q.ifrate / c.rate
7:          if cl не активен then
8:              c.sn ← Q.cycle + vl
9:          else
10:             c.sn ← c.sn + vl
11:         end if
12:         pkt.sn ← c.sn
13:     else
14:         DROP(Q, pkt)
15:     end if
16: end function
```

Сначала нужно классифицировать пакет в очередь. В строке 2 функция классификации определяет очередь, которой соответствует пакет, с помощью

заданных фильтров и возвращает указатель на класс. В строке 3 проверяем, достигла ли очередь предела, отбрасываем его, если достигла (в строке 4), иначе пытаемся добавить его в очередь (в строке 5). Если добавить пакет в очередь удалось, вычисляем его порядковый номер. Для начала вычисляется виртуальная длина пакета в строке 6. Далее в зависимости от того, была ли очередь активна, высчитываем текущий порядковый номер пакета (строки 9 и 11). Если же добавить в очердь не получилось, отбрасываем пакет (строка 14).

Блок-схема алгоритма приведена на рисунке 12.

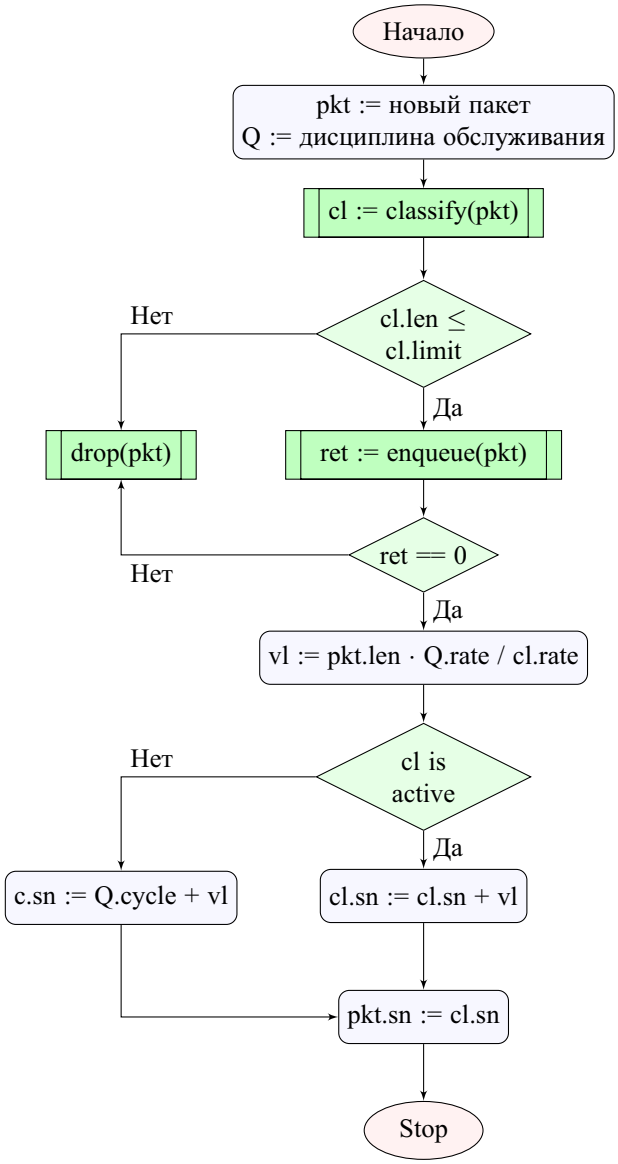


Рисунок 12 — Блок-схема алгоритма добавления пакета в очередь.

2.4.3 Удаление пакета из очереди

Функция удаления пакета из очереди непосредственно реализует планировщик WFQ.

```
1:  function DEQUEUE(Q)
2:      C ← FIND_MIN(Q)
3:      if C is null then
4:          return null
5:      end if
6:      pkt ← C.QUEUE.DEQUEUE(C.queue)
7:      if c.queue.len == 0 then
8:          cl.sn ← 0
9:      end if
10:     if все классы не активны then
11:         Q.sn ← 0
12:     else
13:         Q.sn ← pkt.sn
14:     end if
15:     return pkt
16: end function
```

В первую очередь в строке 2 находим класс с наименьшим порядковым номером пакета в голове очереди. Если класс не был найден, значит все очереди пусты (строка 3), и алгоритм не может вернуть пакет. Иначе достаём пакет из очереди (строка 6). Проверяем, пуста ли очередь, в строке 8. Если пуста, то класс становится неактивным и счётчик сбрасывается (строка 11). В ином случае увеличиваем счётчик циклов всей дисциплины в строке 13. И возвращаем пакет. Блок-схема алгоритма приведена на рисунке 13.

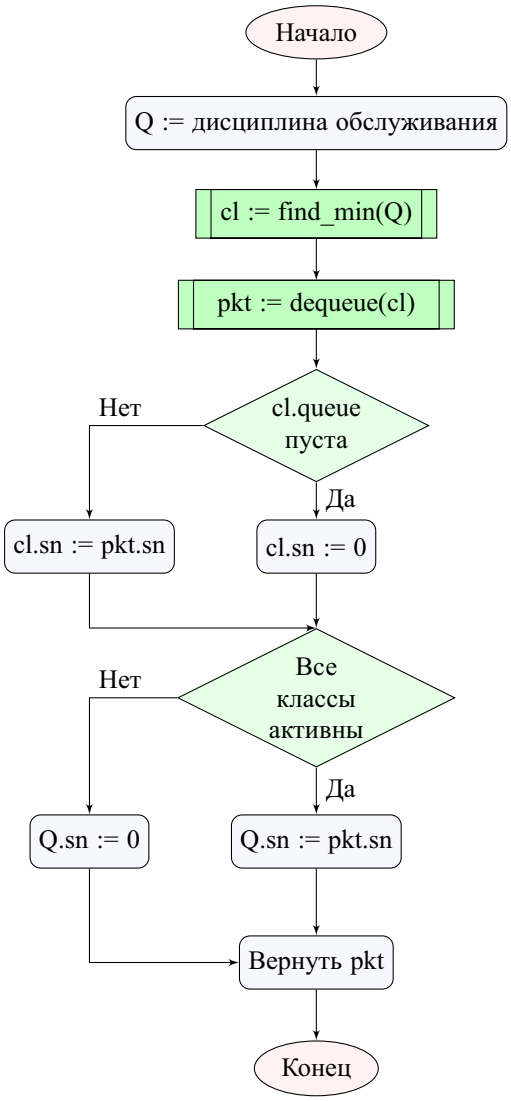


Рисунок 13 — Блок-схема алгоритма удаления пакета из очереди.

3 ТЕСТИРОВАНИЕ РАЗРАБОТАННОГО МОДУЛЯ ДИСЦИПЛИНЫ ОБСЛУЖИВАНИЯ CLASS-BASED WFQ

3.1 Описание тестовой среды

Для тестирования модуля ядра была создана система виртуальных машин на основе системы эмуляции программного обеспечения QEMU. Схема тестовой среды представлена на Рисунке 14. Источником служит узел, от которого исходит трафик; таблицы маршрутизации настроены таким образом, чтобы весь трафик, который должен попасть на узел-цель шёл через промежуточный узел, на котором настроена тестируемая дисциплина обслуживания CBWFQ.

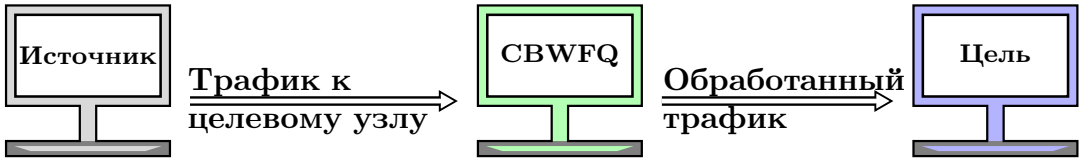


Рисунок 14 — Схема тестовой среды.

В Листинге 1 приведена система настройки дисциплины на промежуточном узле. Переменные окружения TESTPORT1 и TESTPORT2 содержат в себе

```
1  tc qdisc add dev $IFACE root handle 1: cbwfq bandwidth \
2      100Mbps default rate 5Mbps
3  tc class add dev $IFACE parent 1: classid 1:2 cbwfq \
4        rate 25Mbps
5  tc class add dev $IFACE parent 1: classid 1:3 cbwfq \
6        rate 70Mbps
7  tc filter add dev ens4 parent 1:1 protocol ip u32 match \
8          ip dport $TESTPORT1 0xffff flowid 1:2
9  tc filter add dev ens4 parent 1:0 protocol ip u32 match \
10         ip dport $TESTPORT2 0xffff flowid 1:3
```

Листинг 1 — Список команд для конфигурации дисциплины обслуживания CBWFQ.

номера портов, с которых будет отправлен трафик на сервер. При добавлении дисциплины на интерфейс необходимо указать пропускную способность канала и выделенную пропускную способность для класса по умолчанию; размер очереди по умолчанию назначается опционально. При конфигурации классов в третьей и пятой строках происходит назначения пропускной способности для класса (возможно также назначение полосы в процентах от общей пропускной способности). В седьмой и девятой строках происходит назначение фильтров, которые будут направлять трафик с исходнымым портами TESTPORT1 и TESTPORT2 в очередь класса 1:2 и 1:3 соответственно. На один класс можно назначит множество фильтров (Linux предоставляет гибкие возможности по настройке фильтрации); это не контролируется непосредственно дисциплиной и находится в компетенции пользователя.

3.2 Анализ точности выделения канала при конкурирующем трафике

Первый эксперимент заключается в исследовании разделения канала между двумя потоками трафика, приходящими со скоростью, больше скорости канала на промежуточном узле.

С использованием описанной конфигурации и указанных в Листингах 2 и 3 команд произвелось десять испытаний в рамках эксперимента (схема эксперимента представлена на Рисунке 15).

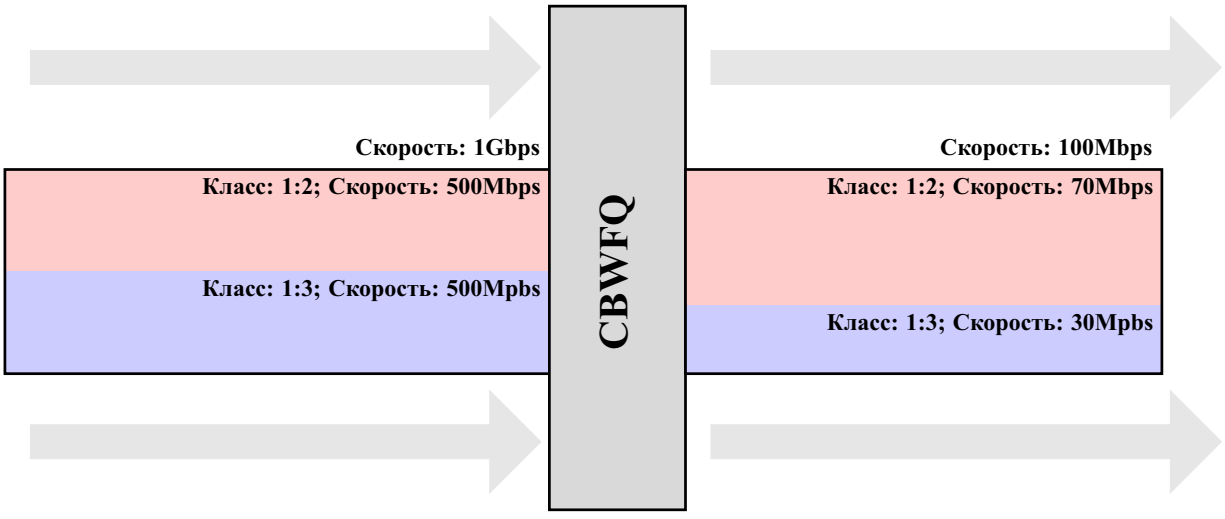


Рисунок 15 — Условная схема эксперимента 1.

В течение экспериментов собирались отчёты от утилиты iperf со стороны

```
iperf3 -c $SERVERIP -p $TESRPORT1 -b 500M -u -t 90
iperf3 -c $SERVERIP -p $TESRPORT2 -b 500M -u -t 90
```

Листинг 2 — Команда iperf на узле-источнике (клиентская сторона).

```
iperf3 -s -p $TESTPOR1 --logfile class2"$EXPNUM".log
iperf3 -s -p $TESTPOR2 --logfile class3"$EXPNUM".log
```

Листинг 3 — Команда iperf на узле-цели (серверная сторона).

сервера в течение 90-та секунд. Отчёты представляют собой таблицу с полями: временной интервал (в секундах), количество переданных данных и пропускная способность. На каждый временной интервал таблица содержит описанную информацию для двух потоков.

Результаты эксперимента можно наблюдать на Рисунке 16.

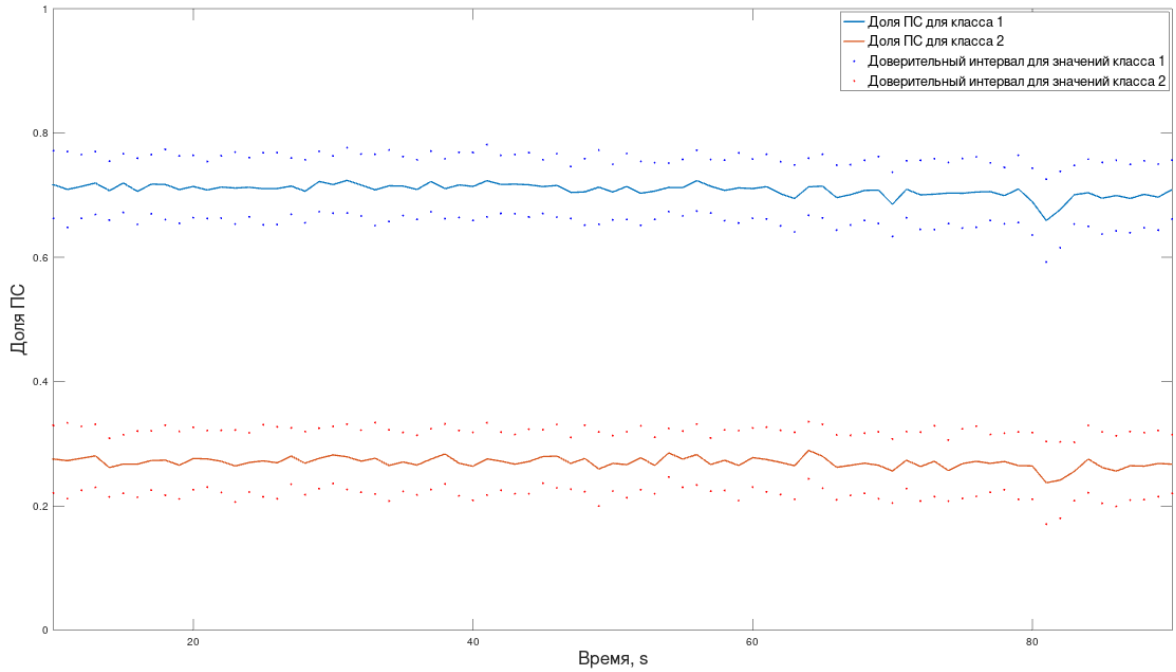


Рисунок 16 — График распределения доли пропускной способности (ПС) по типам трафика в течение времени. Среднее значение процента ПС для класса 1: $71 \pm 3\%$ $(P = 0.95)$. Среднее значение процента ПС для класса 2: $27 \pm 2\%$ $(P = 0.95)$.

Доля пропускной способности для первого и второго класса соответствует минимальной сконфигурированной доле, выделенной этим классам (или их весу $\frac{70\text{Mbps}}{100\text{Mbps}} = 0.7$ и $\frac{25\text{Mbps}}{100\text{Mbps}} = 0.25$ соответственно). Эти доли означают

минимальное количество, выделяемое классу. В данном эксперименте классы получили пропускную способность больше сконфигурированной из-за того, что часть неутилизированной пропускной способности класса по умолчанию распределилась между ними.

3.3 Анализ точности выделения канала при независимом трафике

Второй эксперимент заключается в исследовании разделения канала между двумя потоками трафика, приходящими со скоростью, суммарно меньше скорости канала и соответствующие конфигурационным параметрам.

С использованием описанной конфигурации и указанных в Листингах 4 и 5 команд произвелось десять испытаний в рамках эксперимента (схема

```
iperf3 -c $SERVERIP -p $TESRPORT1 -b 100M -u -t 90
iperf3 -c $SERVERIP -p $TESRPORT2 -b 5M -u -t 90
```

Листинг 4 — Команда iperf на узле-источнике (клиентская сторона).

```
iperf3 -s -p $TESTPOR1 --logfile class2"$EXPNUM".log
iperf3 -s -p $TESTPOR2 --logfile class3"$EXPNUM".log
```

Листинг 5 — Команда iperf на узле-цели (серверная сторона).

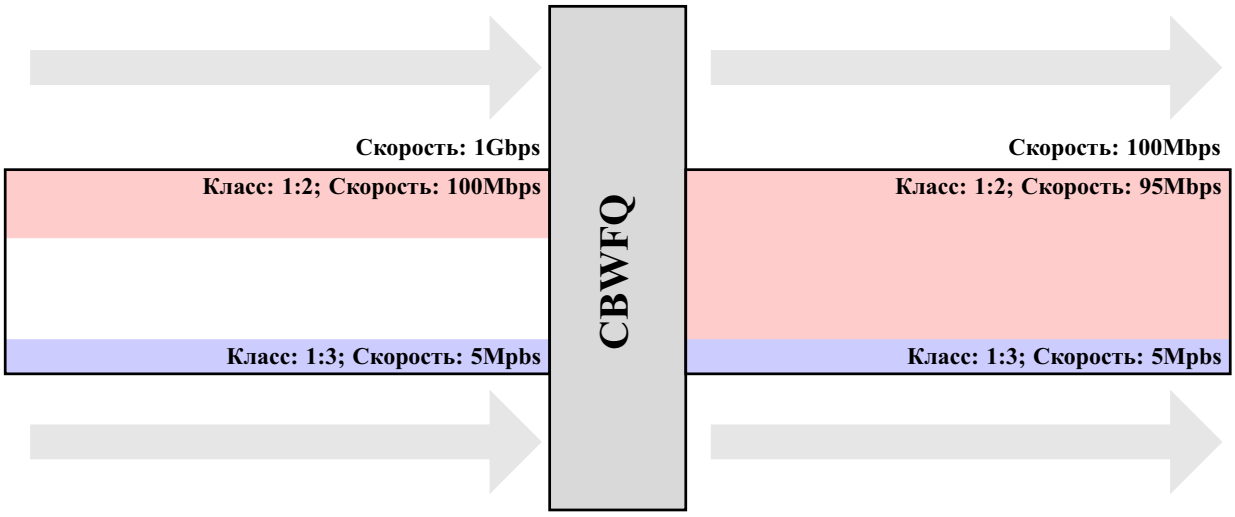


Рисунок 17 — Условная схема эксперимента 2.

В течение экспериментов собирались отчёты от утилиты iperf со стороны сервера в течение 90-та секунд. Отчёты имеют тот же формат, что и описанный в прошлом эксперименте.

Результаты эксперимента можно наблюдать на Рисунке 18.

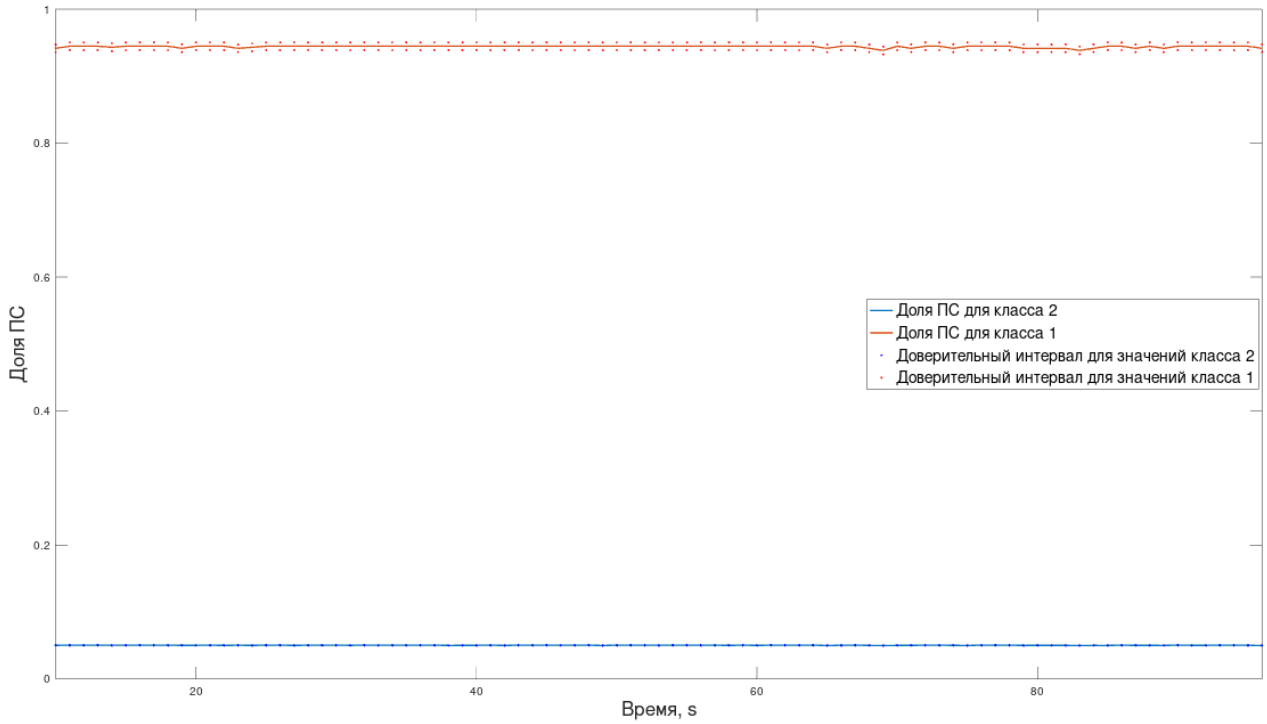


Рисунок 18 — График распределения доли пропускной способности (ПС) по типам трафика в течение времени. Среднее значение процента ПС для класса 1: $94 \pm 0.5\%$ $(P = 0.95)$. Среднее значение процента ПС для класса 2: $5 \pm 0.07\%$ $(P = 0.95)$

В данном случае второй класс получает всю требуемую ему пропускную способность; в этом факте и состоит отсутствие конкуренции: второй поток не пытается занять больше, чем ему назначенно. Поток первого класса занимает оставшуюся свободную часть канала.

ЗАКЛЮЧЕНИЕ

В результате выполнения работы были достигнуты следующие цели:

- проведён сравнительный анализ дисциплин обслуживания PQ, CBQ, HTB, HFSC, FWFQ, CBWFQ; дано краткое описание алгоритмов и приведены их слабые и сильные стороны;
- проведено исследование архитектуры подсистемы контроля качества обслуживания ядра Linux, описаны механизмы работы подсистемы;
- реализован и протестирован модуль дисциплины обслуживания Class-Based WFQ для ядра Linux;
- реализован модуль для утилиты tc для взаимодействия с модулем дисциплины обслуживания.

В дальнейшем работу можно развить в следующих направлениях.

1. Реализация взвешеннего алгоритм раннего обнаружения (WRED) для возможности конфигурации политики отбрасывания для модуля CBWFQ;
2. Доработка работы до дисциплины Low-Latency Queuing (LLQ), которая совершенствует дисциплину CBWFQ с помощью добавления приоритетных очередей, использующиеся для чувствительного к задержкам трафика.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Куклина, М.Д. Исследование и реализация взвешенного алгоритма честного обслуживания на основе классов / М.Д. Куклина, Д.Н. Шинкарук // Сборник тезисов докладов конгресса молодых ученых. Электронное издание [Электронный ресурс]. - Режим доступа: http://openbooks.ifmo.ru/ru/file/7367/7367.pdf.

[2] Sameer Seth. TCP/IP Architecture, Design, and Implementation in Linux. / Sameer Seth, M. Ajaykumar Venkatesulu. // – Wiley-IEEE Computer Society Press, - 2008. - 772 с.

[3] Вегешна Ш. Качество обслуживания в сетях IP. / Ш. Вегешна. – М.: Издательский дом Вильямс, 2003. - 368 с.

[4] Алиев, Т.И. Основы моделирования дискретных систем / Т.И. Алиев. – Спб.: СПбГУ ИТМО. – 2009. – 363 с.

[5] Песин, И. Повесть о Linux и управлении трафиком. [Электронный ресурс] / И. Песин // – 2003. – Режим доступа: http://computerlib.narod.ru/html/traffic.htm (дата обращения 10.04.2018).

[6] Bert Hubert. Linux Advanced Routing and Traffic Control. [Электронный ресурс] // – 2012. – Режим доступа: lartc.org/lartc.pdf (дата обращения 19.03.2018).

[7] Davide Astuti. Packet handling. // Seminar on Transport of Multimedia Streams in Wireless Internet. – 2003.

[8] Alexey N. Kuznetsov. tc-prio (8), . [Электронный ресурс] // Linux Man Pages – Режим доступа: https://www.systutorials.com/docs/linux/man/8-tc-prio/ (дата обращения 10.04.2018).

[9] Chuck Semeria. Supporting differentiated service classes. [Электронный ресурс] // – 2001. Режим доступа: https://pdfs.semanticscholar.org/dd1f/27c4b1e0b5395d67520e65737c9835a7bced.pdf (дата обращения 30.03.2018).

[10] Alexey N. Kuznetsov. tc-cbq (8), . [Электронный ресурс] // Linux Man Pages – Режим доступа: https://www.systutorials.com/docs/linux/man/8-tc-cbq/ (дата обращения 10.04.2018).

[11] Sally Floyd. Link-sharing and resource management models for packet networks. /Sally Floyd, Van Jacobson // IEEE/ACM Transactions on Networking, Vol. 3 No. 4. – 1995.

[12] Martin Devera. Htb linux queuing discipline manual – user guide, 2002. [Электронный ресурс] // Режим доступа: http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm (дата обращения 21.04.2018).

[13] Martin Devera. tc-htb (8). [Электронный ресурс] // Linux Man Pages – Режим доступа: https://www.systutorials.com/docs/linux/man/8-tc-htb/ (дата обращения 11.04.2018).

[14] Ion Stoica. A hierarchical fair service curve algorithm for link-sharing, real-time and priority service, 1997. [Электронный ресурс] / Ion Stoica, Hui Zhang, T.S. Eugene Ng // Режим доступа: https://www.trash.net/~kaber/hfsc/SIGCOM97.pdf (дата обращения 13.03.2018).

[15] Michal Soltys. tc-hfsc (7). [Электронный ресурс] // Linux Man Pages – Режим доступа: https://www.systutorials.com/docs/linux/man/7-tc-hfsc/ (дата обращения 11.04.2018).

[16] Linux-tc-notes. Notes on the Linux Traffic Control Engine. [Электронный ресурс] – 2014. – Режим доступа: http://linux-tc-notes.sourceforge.net/tc/doc/sch_hfsc.txt (дата обращения 23.03.2018).

[17] Cisco IOS Quality of Service Solutions Configuration Guide, Release 12.2 [Электронный ресурс] – 2009. – Режим доступа: https://www.cisco.com/c/en/us/td/docs/ios/qos/configuration/guide/12_2sr/qos_12_2sr_book.pdf (дата обращения 18.04.2018).

[18] Abhay K. Parekh. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. / Abhay K. Parekh, Robert G. Gallager // IEEE/ACM Transactions on Networking, Vol. 1, No. 3 – 1993.

[19] Quality of Service, Part 10 – Weighted Fair Queuing. [Электронный ресурс] // – 2010. – Режим доступа: http://blog.globalknowledge.com/2010/02/12/quality-of-service-part-10-weighted-fair-queuing/ (дата обращения 04.03.2018).

[20] Aaron Blachunas. Qos and queueing. [Электронный ресурс] // – 2010. – Режим доступа: http://www.routeralley.com/guides/qos_queuing.pdf (дата обращения 09.04.2018).

СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

– ДО — Дисциплина обслуживания.
– ПС — пропускная способность.
– CBWFQ — Class Based Weighted Fail Queueing.
– CBQ — Class Based Queueing.
– ED/AD — Early-Detection/Aggressive-Detection.
– HTB — Hierarchical Token Bucket.
– HFSC — Hierarchical Fair-Service Curve.
– PQ — Priority Queueing.
– RT/LS — Real-Time/Link-Sharing.
– RR — Round Robin.
– TD — Tail-Drop.
– WFQ — Weighted Fair Queueing.
– WRR — Weighted Round Robin.
– WRED — Weighted Random Early Detection.

ПРИЛОЖЕНИЕ А

```
937,987d936
< enum {
<       TCA_CBWFQ_UNSPEC,
<       TCA_CBWFQ_PARAMS,
<       TCA_CBWFQ_INIT,
<       __TCA_CBWFQ_MAX
< };
< #define TCA_CBWFQ_MAX (__TCA_CBWFQ_MAX - 1)
<
< enum cbwfq_rate_type {
<       TCA_CBWFQ_RT_BYTE,
<       TCA_CBWFQ_RT_PERCENT
< };
<
< struct tc_cbwfq_glob {
<       __u32 cbwfq_gl_default_limit;
<       __u32 cbwfq_gl_default_rate;
<       __u32 cbwfq_gl_total_rate;
<       enum cbwfq_rate_type cbwfq_gl_rate_type;
< };
<
< struct tc_cbwfq_copt {
<       __u32 cbwfq_cl_rate;
<       __u32 cbwfq_cl_limit;
<       enum cbwfq_rate_type cbwfq_cl_rate_type;
< };
<
```

Листинг 6 — Патч для заголовочного файла pkt_sched.h.

ПРИЛОЖЕНИЕ Б

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <syslog.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>

#include "utils.h"
#include "tc_util.h"

static void explain(void)
{
    fprintf(stderr,
"Usage: ... qdisc add .. cbwfq bandwidth B default rate R [limit L] \n"
"\tbandwidth          bandwidth of the link (Mbps, Kbps, bps)\n"
"\tdefault            configuration for default class (see desciption below)\
   n"
"... class add ... cbwfq rate R [limit L]\n"
"\trate R [percent]   rate of the class in Kbit; use 'percent' to declarein
   percent\n"
"\tlimit              max queue length (in packets)\n"
    );
}

static void explain1(char *arg)
{
    fprintf(stderr, "Illegal \"%s\"\n", arg);
    explain();
}

static int cbwfq_parse_opt(struct qdisc_util *qu, int argc, char **argv,
                           struct nlmsghdr *n)
{
    struct tc_cbwfq_glob opt;
    struct rtattr *tail;

    memset(&opt, 0, sizeof(opt));
    while (argc > 0) {
        if (matches(*argv, "default") == 0) {
            NEXT_ARG();
            if (matches(*argv, "rate") == 0) {
                NEXT_ARG();
                if (get_rate(&opt.cbwfq_gl_default_rate, *argv)) {
                    explain1("rate");
                    return -1;
                }
                argv++; argc--;
                if (argc <= 0) {
                        break;
                }
                if (matches(*argv, "percent") == 0) {
                    opt.cbwfq_gl_rate_type = TCA_CBWFQ_RT_PERCENT;
```

```
                    argv--;
                    if (get_u32(&opt.cbwfq_gl_default_rate, *argv, 10)) {
                        explain1("percent");
                        return -1;
                    }
                    argv++;
                } else {
                    opt.cbwfq_gl_rate_type = TCA_CBWFQ_RT_BYTE;
                }
            }
            if (matches(*argv, "limit") == 0) {
                NEXT_ARG();
                if (get_u32(&opt.cbwfq_gl_default_limit, *argv, 10)) {
                    explain1("limit");
                    return -1;
                }
            } else {
                fprintf(stderr, "Unknown default parameter: \"%s\".\n", *argv
                    );
                explain();
                return -1;
            }
        } else if (matches(*argv, "bandwidth") == 0) {
                NEXT_ARG();
                if (get_rate(&opt.cbwfq_gl_total_rate, *argv)) {
                    explain1("bandwidth");
                    return -1;
                }
        } else {
            fprintf(stderr, "What is \"%s\"?\n", *argv);
            explain();
            return -1;
        }
        argc--; argv++;
    }

    if (opt.cbwfq_gl_total_rate <= 0) {
        fprintf(stderr, "Bandwidth must be set!\n");
        return -1;
    }

    if (opt.cbwfq_gl_default_rate <= 0) {
        fprintf(stderr, "Default rate must be set!\n");
        return -1;
    }

    tail = NLMSG_TAIL(n);
    addattr_l(n, 1024, TCA_OPTIONS, NULL, 0);
    addattr_l(n, 2024, TCA_CBWFQ_INIT, &opt, NLMSG_ALIGN(sizeof(opt)));
    tail->rta_len = (void *) NLMSG_TAIL(n) - (void *) tail;
    return 0;
}

static int cbwfq_parse_class_opt(struct qdisc_util *qu, int argc, char **argv
    ,
                                 struct nlmsghdr *n)
{
    struct tc_cbwfq_copt opt;
    struct rtattr *tail;
```

```
    memset(&opt, 0, sizeof(opt));
    while (argc > 0) {
        if (matches(*argv, "rate") == 0) {
            NEXT_ARG();
            opt.cbwfq_cl_rate_type = TCA_CBWFQ_RT_BYTE;
            if (get_rate(&opt.cbwfq_cl_rate, *argv)) {
                explain1("rate");
                return -1;
            }

            argv++; argc--;
            if (argc <= 0) {
                    break;
            }

            if (matches(*argv, "percent") == 0) {
                opt.cbwfq_cl_rate_type = TCA_CBWFQ_RT_PERCENT;
                argv--;
                if (get_u32(&opt.cbwfq_cl_rate, *argv, 10)) {
                    explain1("percent");
                    return -1;
                }
                argv++;
            } else {
                fprintf(stderr, "What is \"%s\"?\n", *argv);
                explain();
                return -1;
            }
        } else if (matches(*argv, "limit") == 0) {
            NEXT_ARG();
            if (get_u32(&opt.cbwfq_cl_limit, *argv, 10)) {
                explain1("limit");
                return -1;
            }
        } else {
            fprintf(stderr, "What is \"%s\"?\n", *argv);
            explain();
            return -1;
        }
        argc--; argv++;
    }


    if (opt.cbwfq_cl_rate <= 0) {
        fprintf(stderr, "Rate must be set!\n");
        explain();
        return -1;
    }

    tail = NLMSG_TAIL(n);
    addattr_l(n, 1024, TCA_OPTIONS, NULL, 0);
    addattr_l(n, 1024, TCA_CBWFQ_PARAMS, &opt, sizeof(opt));
    tail->rta_len = (void *) NLMSG_TAIL(n) - (void *) tail;
    return 0;
}

int cbwfq_print_opt(struct qdisc_util *qu, FILE *f, struct rtattr *opt)
{
    struct tc_cbwfq_glob *qopt = NULL;
    struct rtattr *tb[TCA_CBWFQ_MAX+1];
```

```
    if (opt == NULL) {
        return 0;
    }

    if (parse_rtattr_nested(tb, TCA_CBWFQ_MAX, opt)) {
        return -1;
    }

    if (tb[TCA_CBWFQ_INIT] == NULL) {
        return -1;
    }

    if (RTA_PAYLOAD(tb[TCA_CBWFQ_INIT]) < sizeof(*opt)) {
        fprintf(stderr, "qdisc opt is too short\n");
    } else {
        qopt = RTA_DATA(tb[TCA_CBWFQ_INIT]);
    }

    if (qopt != NULL) {
        fprintf(f, "total rate %d ", qopt->cbwfq_gl_total_rate);
    }
    return 0;
}

static int cbwfq_print_copt(struct qdisc_util *qu, FILE *f, struct rtattr *
    opt)
{
    struct rtattr *tb[TCA_CBWFQ_MAX+1];
    struct tc_cbwfq_copt *copt = NULL;

    if (opt == NULL)
        return 0;

    if (parse_rtattr_nested(tb, TCA_CBWFQ_MAX, opt))
        return -1;

    if (tb[TCA_CBWFQ_PARAMS] != NULL) {
        if (RTA_PAYLOAD(tb[TCA_CBWFQ_PARAMS]) < sizeof(*opt))
            fprintf(stderr, "CBWFQ: class opt is too short\n");
        else
            copt = RTA_DATA(tb[TCA_CBWFQ_PARAMS]);
    }

    if (copt) {
        fprintf(f, "limit %d rate %d ", copt->cbwfq_cl_limit,
                                        copt->cbwfq_cl_rate);
    }

    return 0;
}

struct qdisc_util cbwfq_qdisc_util = {
    .id           = "cbwfq",
    .parse_copt   = cbwfq_parse_class_opt,
    .parse_qopt   = cbwfq_parse_opt,
    .print_qopt   = cbwfq_print_opt,
    .print_copt   = cbwfq_print_copt,
};
```

Листинг 7 — Модуль CBWFQ для утилиты tc.

ПРИЛОЖЕНИЕ В

```c
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/string.h>
#include <linux/errno.h>
#include <linux/skbuff.h>
#include <net/netlink.h>
#include <net/pkt_sched.h>
#include <net/pkt_cls.h>

#include <linux/list.h>

#define MAX(a, b) ((a) > (b) ? (a) : (b))

#define DEFAULT_CL_ID     65537

/**
 * cbwfq_class -- class description
 * @common      Common qdisc data. Used in hash-table.
 * @queue       Class queue.
 *
 * @limit       Max amount of packets.
 * @rate      Assigned rate.
 *
 * @cl_sn     Sequence number of the last enqueued packet.
 *
 * @is_active  Set if class is in active state (transmit packets).
 */
struct cbwfq_class {
    struct Qdisc_class_common common;
    struct Qdisc *queue;

    u64 limit;
    u64 rate;

    u64 cl_sn;

    bool is_active;
};

/**
 * cbwfq_sched_data -- scheduler data
 *
 * @clhash  Hash table of classes.
 *
 * @filter_list       List of attached filters.
 * @block             Field used for filters to work.
 *
 * @default_queue     Default class with the default queue.
 *
 * @ifrate          Total rate of the link.
 * @active_rate     Rate of all acitve classes. Used to determine idle.
 *
 * @sch_sn        Sequence number of the last dequeued packet; cycle number.
```

```
 */
struct cbwfq_sched_data {
    struct Qdisc_class_hash clhash;

    struct tcf_proto __rcu *filter_list;
    struct tcf_block *block;

    struct cbwfq_class *default_queue;

    enum cbwfq_rate_type rtype;
    u64 ifrate;

    u32 active_rate;

    u64 sch_sn;
};


/* For parsing netlink messages. */
static const struct nla_policy cbwfq_policy[TCA_CBWFQ_MAX + 1] = {
    [TCA_CBWFQ_PARAMS]  = { .len = sizeof(struct tc_cbwfq_copt) },
    [TCA_CBWFQ_INIT]    = { .len = sizeof(struct tc_cbwfq_glob) },
};


/**
 * Add class to the hash table.
 *
 * @comment Class is allocated outside.
 */
static void
cbwfq_add_class(struct Qdisc *sch, struct cbwfq_class *cl)
{
    struct cbwfq_sched_data *q = qdisc_priv(sch);
    cl->queue = qdisc_create_dflt(sch->dev_queue,
                                  &pfifo_qdisc_ops, cl->common.classid);
    qdisc_class_hash_insert(&q->clhash, &cl->common);
}

/**
 * Destroy class.
 *
 * @se Free memory.
 */
static void
cbwfq_destroy_class(struct Qdisc *sch, struct cbwfq_class *cl)
{
    struct cbwfq_sched_data *q = qdisc_priv(sch);

    sch_tree_lock(sch);

    qdisc_tree_reduce_backlog(cl->queue, cl->queue->q.qlen,
                      cl->queue->qstats.backlog);
    qdisc_class_hash_remove(&q->clhash, &cl->common);

    sch_tree_unlock(sch);

    if (cl->queue) {
        qdisc_destroy(cl->queue);
    }
```

```
    kfree(cl);
}

/**
 * Find class with given classid.
 */
static inline struct cbwfq_class *
cbwfq_class_lookup(struct cbwfq_sched_data *q, u32 classid)
{
    struct Qdisc_class_common *clc;

    clc = qdisc_class_find(&q->clhash, classid);
    if (clc == NULL)
        return NULL;
    return container_of(clc, struct cbwfq_class, common);
}

/**
 *  Find class with given id and return its address.
 */
static unsigned long
cbwfq_find(struct Qdisc *sch, u32 classid)
{
    struct cbwfq_sched_data *q = qdisc_priv(sch);
  return (unsigned long)cbwfq_class_lookup(q, classid);
}

/**
 * Modify class with given options.
 */
static int
cbwfq_modify_class(struct Qdisc *sch, struct cbwfq_class *cl,
                   struct tc_cbwfq_copt *copt)
{
    struct cbwfq_sched_data *q = qdisc_priv(sch);

    if (copt->cbwfq_cl_limit > 0) {
        cl->limit = copt->cbwfq_cl_limit;
    }

    if (copt->cbwfq_cl_rate_type != q->rtype) {
        PRINT_INFO_ARGS("different rate types.");
        return -EINVAL;
    }
    cl->rate = copt->cbwfq_cl_rate;

    return 0;
}

/**
 * Create new class.
 */
static int
cbwfq_class_create(struct Qdisc *sch, struct tc_cbwfq_copt *copt,
                   unsigned long classid)
{
    struct cbwfq_class *cl;
    struct cbwfq_sched_data *q = qdisc_priv(sch);

    cl = kmalloc( sizeof(struct cbwfq_class), GFP_KERNEL);
```

```
    if (cl == NULL)
        return -ENOMEM;

    cl->common.classid = classid;
    cl->limit      = 1000;
    cl->rate       = 0;
    cl->cl_sn      = 0;
    cl->is_active = false;

    if (cbwfq_modify_class(sch, cl, copt) != 0) {
        kfree(cl);
        return -EINVAL;
    }

    sch_tree_lock(sch);
        cbwfq_add_class(sch, cl);
    sch_tree_unlock(sch);

    return 0;
}

/**
 * Add or change a class by given id.
 *
 * @se Allocated memory.
 */
static int
cbwfq_change_class(struct Qdisc *sch, u32 classid, u32 parentid,
                   struct nlattr **tca, unsigned long *arg)
{
    struct cbwfq_sched_data *q = qdisc_priv(sch);
    struct cbwfq_class *cl;
    struct nlattr *opt = tca[TCA_OPTIONS];
    struct nlattr *tb[TCA_CBWFQ_MAX + 1];
    struct tc_cbwfq_copt *copt;
    int err;
    int p_maj   = TC_H_MAJ(parentid) >> 16;
    int cid_maj = TC_H_MAJ(classid) >> 16;

    /* Both have to be 1, because there's no class heirarchy. */
    if (p_maj != 1 || cid_maj != 1)
        return -EINVAL;

    if (opt == NULL) {
        return -EINVAL;
    }

    err = nla_parse_nested(tb, TCA_CBWFQ_MAX, opt, cbwfq_policy, NULL);
    if (err < 0) {
        return err;
    }

    if (tb[TCA_CBWFQ_PARAMS] == NULL) {
        return -EINVAL;
    }
    copt = nla_data(tb[TCA_CBWFQ_PARAMS]);

    cl = cbwfq_class_lookup(q, classid);
    if (cl != NULL) {
        return cbwfq_modify_class(sch, cl, copt);
```

```
    }
    return cbwfq_class_create(sch, copt, classid, extack);
}

/**
 * Delete class by given id.
 *
 * @se Free memory.
 */
static int
cbwfq_delete_class(struct Qdisc *sch, unsigned long arg)
{
    struct cbwfq_class *cl = (struct cbwfq_class *)arg;

    if (cl == NULL || cl->common.classid == DEFAULT_CL_ID) {
        return -EINVAL;
    }

    cbwfq_destroy_class(sch, cl);
    return 0;
}

/**
 * Classify the given packet to a class.
 */
static struct cbwfq_class *
cbwfq_classify(struct sk_buff *skb, struct Qdisc *sch, int *qerr)
{
    struct cbwfq_sched_data *q = qdisc_priv(sch);
    struct cbwfq_class *cl;
    struct tcf_result res;
    struct tcf_proto *fl;
    int err;
    u32 classid = TC_H_MAKE(1 << 16, 1);

    *qerr = NET_XMIT_SUCCESS | __NET_XMIT_BYPASS;
    if (TC_H_MAJ(skb->priority) != sch->handle) {
        fl = rcu_dereference_bh(q->filter_list);
        err = tcf_classify(skb, fl, &res, false);

        if (!fl || err < 0) {
            return q->default_queue;
        }

#ifdef CONFIG_NET_CLS_ACT
        switch (err) {
            case TC_ACT_STOLEN:
            case TC_ACT_QUEUED:
            case TC_ACT_TRAP:
                *qerr = NET_XMIT_SUCCESS | __NET_XMIT_STOLEN;
                /* fall through */
            case TC_ACT_SHOT:
                return NULL;
        }
#endif

        classid = res.classid;
    }

    return cbwfq_class_lookup(q, classid);
```

```
}

/**
 * Enqueue packet to the queue.
 */
static int
cbwfq_enqueue(struct sk_buff *skb, struct Qdisc *sch, struct sk_buff **
    to_free)
{
    struct cbwfq_sched_data *q = qdisc_priv(sch);
    struct cbwfq_class *cl;
    struct Qdisc *qdisc;
    int ret;

    cl = cbwfq_classify(skb, sch, &ret);
    if (cl == NULL || cl->queue == NULL) {
        if (ret & __NET_XMIT_BYPASS)
            qdisc_qstats_drop(sch);
        __qdisc_drop(skb, to_free);
        return ret;
    }

    if (cl->queue->q.qlen >= cl->limit) {
        if (net_xmit_drop_count(ret)) {
            qdisc_qstats_drop(sch);
        }
        return qdisc_drop(skb, sch, to_free);
    }

    qdisc = cl->queue;
    ret = qdisc_enqueue(skb, qdisc, to_free);
    if (ret == NET_XMIT_SUCCESS) {
        u32 virtual_len = qdisc_pkt_len(skb) * (q->ifrate / cl->rate);
        if (!cl->is_active) {
           cl->cl_sn = q->sch_sn + virtual_len;
           cl->is_active = true;
           q->active_rate += cl->rate;
        } else {
            cl->cl_sn += virtual_len;
        }
        skb->tstamp = cl->cl_sn;

        sch->q.qlen++;
        qdisc_qstats_backlog_inc(sch, skb);
        qdisc_qstats_backlog_inc(cl->queue, skb);
        return NET_XMIT_SUCCESS;
    }

    if (net_xmit_drop_count(ret)) {
        qdisc_qstats_drop(sch);
        qdisc_qstats_drop(cl->queue);
    }
    return ret;
}

/**
 * Find minimum class with minimum sequence number.
 */
static struct cbwfq_class *
cbwfq_find_min(struct cbwfq_sched_data *q)
```

```
{
    struct cbwfq_class *it, *cl = NULL;
    ktime_t ft = KTIME_MAX;
    int i;

    for (i = 0; i < q->clhash.hashsize; i++) {
        hlist_for_each_entry(it, &q->clhash.hash[i], common.hnode) {
            if (it->is_active) {
                struct sk_buff *skb = it->queue->ops->peek(it->queue);
                if (ft > skb->tstamp) {
                    cl = it;
                    ft = skb->tstamp;
                }
            }
        }
    }
    return cl;
}

/**
 * Return packet without deleting it from a queue.
 */
static struct sk_buff *
cbwfq_peek(struct Qdisc *sch)
{
    struct cbwfq_sched_data *q = qdisc_priv(sch);
    struct cbwfq_class *cl = NULL;

    cl = cbwfq_find_min(q);
    if (cl == NULL) {
        return NULL;
    }
    return cl->queue->ops->peek(cl->queue);
}

/**
 * Dequeue packet.
 */
static struct sk_buff *
cbwfq_dequeue(struct Qdisc *sch)
{
    struct cbwfq_sched_data *q = qdisc_priv(sch);
    struct cbwfq_class *cl = NULL;
    struct sk_buff *skb;

    cl = cbwfq_find_min(q);
    if (cl == NULL) {
        return NULL;
    }

    skb = cl->queue->ops->dequeue(cl->queue);
    if (skb == NULL) {
        return NULL;
    }

    qdisc_bstats_update(sch, skb);
    qdisc_qstats_backlog_dec(sch, skb);
    qdisc_qstats_backlog_dec(cl->queue, skb);
    sch->q.qlen--;
```

```
    if (cl->queue->q.qlen == 0) {
        cl->is_active = false;
        cl->cl_sn     = 0;
        q->active_rate -= cl->rate;
    }

    if (q->active_rate == 0) {
        q->sch_sn = 0;
    } else {
        q->sch_sn = skb->tstamp;
    }
    return skb;
}

/**
 * Reset qdisc.
 */
static void
cbwfq_reset(struct Qdisc *sch)
{
    int i;
    struct cbwfq_sched_data *q = qdisc_priv(sch);
    struct cbwfq_class *it;

    q->sch_sn = 0;
    for (i = 0; i < q->clhash.hashsize; i++) {
        hlist_for_each_entry(it, &q->clhash.hash[i], common.hnode) {
            cl->cl_sn = 0;
            cl->is_active = false;
            qdisc_reset(it->queue);
        }
    }
    sch->qstats.backlog = 0;
    sch->q.qlen = 0;
}

/**
 * Destoy qdisc.
 */
static void
cbwfq_destroy(struct Qdisc *sch)
{
    int i;
    struct cbwfq_sched_data *q = qdisc_priv(sch);
    struct cbwfq_class *it;
    struct hlist_node *next;

    tcf_block_put(q->block);

    for (i = 0; i < q->clhash.hashsize; i++) {
        hlist_for_each_entry_safe(it, next, &q->clhash.hash[i], common.hnode)
            {
            if (it != NULL) {
                cbwfq_destroy_class(sch, it);
            }
        }
    }
  qdisc_watchdog_cancel(&q->watchdog);
    qdisc_class_hash_destroy(&q->clhash);
}
```

```
/**
 * Change qdisc configuration.
 */
static int
cbwfq_change(struct Qdisc *sch, struct nlattr *opt)
{
    struct cbwfq_sched_data *q = qdisc_priv(sch);
    struct tc_cbwfq_glob *qopt;
    struct nlattr *tb[TCA_CBWFQ_MAX + 1];
    int err;

    if (opt == NULL) {
        return -EINVAL;
    }

    err = nla_parse_nested(tb, TCA_CBWFQ_MAX, opt, cbwfq_policy, NULL);
    if (err < 0) {
        return err;
    }

    if (tb[TCA_CBWFQ_INIT] == NULL) {
        return -EINVAL;
    }

    qopt = nla_data(tb[TCA_CBWFQ_INIT]);

    sch_tree_lock(sch);

    if (qopt->cbwfq_gl_default_limit > 0) {
        q->default_queue->limit = qopt->cbwfq_gl_default_limit;
    }

    sch_tree_unlock(sch);
    return 0;
}

/**
 * Initilize new instance of qdisc.
 */
static int
cbwfq_init(struct Qdisc *sch, struct nlattr *opt)
{
    struct cbwfq_sched_data *q = qdisc_priv(sch);
    struct cbwfq_class *cl;
    struct tc_cbwfq_glob *qopt;
    struct nlattr *tb[TCA_CBWFQ_MAX + 1];
    int err;

    if (!opt)
        return -EINVAL;

    /* Init filter system. */
    err = tcf_block_get(&q->block, &q->filter_list, sch, extack);
    if (err)
        return err;

    /* Init hash table for class storing. */
    err = qdisc_class_hash_init(&q->clhash);
    if (err < 0)
```

```
        return err;

    q->active_rate = 0;
    q->sch_sn      = 0;

    /* Init default queue. */
    cl = kmalloc( sizeof(struct cbwfq_class), GFP_KERNEL);
    if (cl == NULL) {
        return -ENOMEM;
    }
    q->default_queue = cl;

    /* Set classid for default class. */
    cl->common.classid = TC_H_MAKE(1 << 16, 1);
    cl->limit   = 1024;
    cl->rate  = 0;
    cl->is_active = false;
    cl->cl_sn = 0;

    err = nla_parse_nested(tb, TCA_CBWFQ_MAX, opt, cbwfq_policy, NULL);
    if (err < 0)
        return err;

    qopt = nla_data(tb[TCA_CBWFQ_INIT]);

    if (qopt->cbwfq_gl_default_limit != 0) {
        cl->limit = qopt->cbwfq_gl_default_limit;
    }

    if (qopt->cbwfq_gl_rate_type == TCA_CBWFQ_RT_BYTE) {
        q->rtype  = TCA_CBWFQ_RT_BYTE;
        q->ifrate = qopt->cbwfq_gl_total_rate;
        cl->rate  = qopt->cbwfq_gl_default_rate;
    } else {
        q->rtype  = TCA_CBWFQ_RT_PERCENT;
        q->ifrate = 100;
        cl->rate  = qopt->cbwfq_gl_default_rate;
    }

    sch_tree_lock(sch);
    cbwfq_add_class(sch, cl, extack);
    sch_tree_unlock(sch);
  qdisc_watchdog_init(&q->watchdog, sch);
    return 0;
}

/**
 * Dump qdisc configuration.
 */
static int
cbwfq_dump(struct Qdisc *sch, struct sk_buff *skb)
{
    struct cbwfq_sched_data *q = qdisc_priv(sch);
    unsigned char *b = skb_tail_pointer(skb);
    struct tc_cbwfq_glob opt;
    struct nlattr *nest;

    memset(&opt, 0, sizeof(opt));
    opt.cbwfq_gl_total_rate = q->ifrate;
```

```
    nest = nla_nest_start(skb, TCA_OPTIONS);
    if (nest == NULL)
        goto nla_put_failure;

    if (nla_put(skb, TCA_CBWFQ_INIT, sizeof(opt), &opt))
        goto nla_put_failure;

    return nla_nest_end(skb, nest);

nla_put_failure:
    nlmsg_trim(skb, b);
    return -1;
}

/**
 * Dump class configuration.
 */
static int
cbwfq_dump_class(struct Qdisc *sch, unsigned long cl,
                 struct sk_buff *skb, struct tcmsg *tcm)
{
    struct cbwfq_class *c = (struct cbwfq_class *)cl;
    struct nlattr *nest;
    struct tc_cbwfq_copt opt;

    if (c == NULL) {
        return -1;
    }

    tcm->tcm_handle = c->common.classid;
    tcm->tcm_info = c->queue->handle;

    nest = nla_nest_start(skb, TCA_OPTIONS);
    if (nest == NULL)
        goto failure;

    memset(&opt, 0, sizeof(opt));
    opt.cbwfq_cl_limit  = c->limit;
    opt.cbwfq_cl_rate = c->rate;

    if (nla_put(skb, TCA_CBWFQ_PARAMS, sizeof(opt), &opt))
        goto failure;

    return nla_nest_end(skb, nest);

failure:
    nla_nest_cancel(skb, nest);
    return -1;
}

/**
 * Dump class statistics.
 */
static int
cbwfq_dump_class_stats(struct Qdisc *sch, unsigned long cl,
                       struct gnet_dump *d)
{
    struct cbwfq_class *c = (struct cbwfq_class*)cl;
    int gs_base, gs_queue;
```

```
    if (c == NULL)
        return -1;

    gs_base = gnet_stats_copy_basic(qdisc_root_sleeping_running(sch),
                                    d, NULL, &c->queue->bstats);
    gs_queue = gnet_stats_copy_queue(d, NULL, &c->queue->qstats,
                                     c->queue->q.qlen);

    return gs_base < 0 || gs_queue < 0 ? -1 : 0;
}

/**
 * Attach a new qdisc to a class and return the prev attached qdisc.
 */
static int
cbwfq_graft(struct Qdisc *sch, unsigned long arg, struct Qdisc *new,
            struct Qdisc **old)
{
    struct cbwfq_sched_data *q = qdisc_priv(sch);
    struct cbwfq_class *cl;

    if (new == NULL)
        new = &noop_qdisc;

    cl = cbwfq_class_lookup(q, arg);
    if (cl) {
        *old = qdisc_replace(sch, new, &cl->queue);
        return 0;
    }
    return -1;
}

/**
 * Returns a pointer to the qdisc of class.
 */
static struct Qdisc *
cbwfq_leaf(struct Qdisc *sch, unsigned long arg)
{
    struct cbwfq_sched_data *q = qdisc_priv(sch);
    struct cbwfq_class *cl = cbwfq_class_lookup(q, arg);

    return cl == NULL? NULL : cl->queue;
}

static unsigned long
cbwfq_bind(struct Qdisc *sch, unsigned long parent, u32 classid)
{
    return cbwfq_find(sch, classid);
}


static void
cbwfq_unbind(struct Qdisc *q, unsigned long cl)
{ }

/**
 * Iterates over all classed of a qdisc.
 */
static void
cbwfq_walk(struct Qdisc *sch, struct qdisc_walker *arg)
```

```
{
    struct cbwfq_sched_data *q = qdisc_priv(sch);
    struct cbwfq_class *cl;
    int h;

    if (arg->stop)
        return;

    for (h = 0; h < q->clhash.hashsize; h++) {
        hlist_for_each_entry(cl, &q->clhash.hash[h], common.hnode) {
            if (arg->count < arg->skip) {
                arg->count++;
                continue;
            }
            if (arg->fn(sch, (unsigned long)cl, arg) < 0) {
                arg->stop = 1;
                break;
            }
            arg->count++;
        }
    }
}

static struct tcf_block *
cbwfq_tcf_block(struct Qdisc *sch, unsigned long cl, struct netlink_ext_ack *
    extack)
{
    return qdisc_priv(sch)->block;
}

static const struct Qdisc_class_ops cbwfq_class_ops = {
    .graft      =   cbwfq_graft,
    .leaf       =   cbwfq_leaf,
    .find       =   cbwfq_find,
    .walk       =   cbwfq_walk,
    .change     =   cbwfq_change_class,
    .delete     =   cbwfq_delete_class,
    .tcf_block  =   cbwfq_tcf_block,
    .bind_tcf   =   cbwfq_bind,
    .unbind_tcf =   cbwfq_unbind,
    .dump       =   cbwfq_dump_class,
    .dump_stats =   cbwfq_dump_class_stats,
};

static struct Qdisc_ops cbwfq_qdisc_ops __read_mostly = {
    /* Points to next Qdisc_ops. */
    .next       =   NULL,
    /* Points to structure that provides a set of functions for
     * a particular class. */
    .cl_ops     =   &cbwfq_class_ops,
    /* Char array contains identity of the qdsic. */
    .id         =   "cbwfq",
    .priv_size  =   sizeof(struct cbwfq_sched_data),

    .enqueue    =   cbwfq_enqueue,
    .dequeue    =   cbwfq_dequeue,
    .peek       =   cbwfq_peek,
    .init       =   cbwfq_init,
    .reset      =   cbwfq_reset,
    .destroy    =   cbwfq_destroy,
```

```
    .change     =   cbwfq_change,
    .dump       =   cbwfq_dump,
    .owner      =   THIS_MODULE,
};

static int __init
cbwfq_module_init(void)
{
    return register_qdisc(&cbwfq_qdisc_ops);
}

static void __exit
cbwfq_module_exit(void)
{
    unregister_qdisc(&cbwfq_qdisc_ops);
}

module_init(cbwfq_module_init)
module_exit(cbwfq_module_exit)

MODULE_LICENSE("GPL");
```

Листинг 8 — Модуль CBWFQ для ядра Linux.