СОДЕРЖАНИЕ

ВВЕДЕНИЕ

1 АНАЛИЗ ДИСЦПЛИН ОБСЛУЖИВАНИЯ ЯДРА LINUX

1.1 Дисциплины обслуживания

Дисциплина обслуживания очередей (ДО) – правило выбора заявок из очереди для обслуживания[1].

Возможно, нужно описать некоторые термины, которые потом будут представлены в таблице.

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 переполнения отбрасываются пакеты из очереди с более низким приоритетом.[2]

Дисциплина используется, чтобы понизить время отклика, когда нет нужды замедлять трафик[3].

В Linux алгоритм реализован в виде дисцплины prio, которая создаёт фиксированное значение очередей обслуживания, управляемые дисциплиной pfast_fifo, и управляет очередями в соответствии с картой приоритетов.[3]

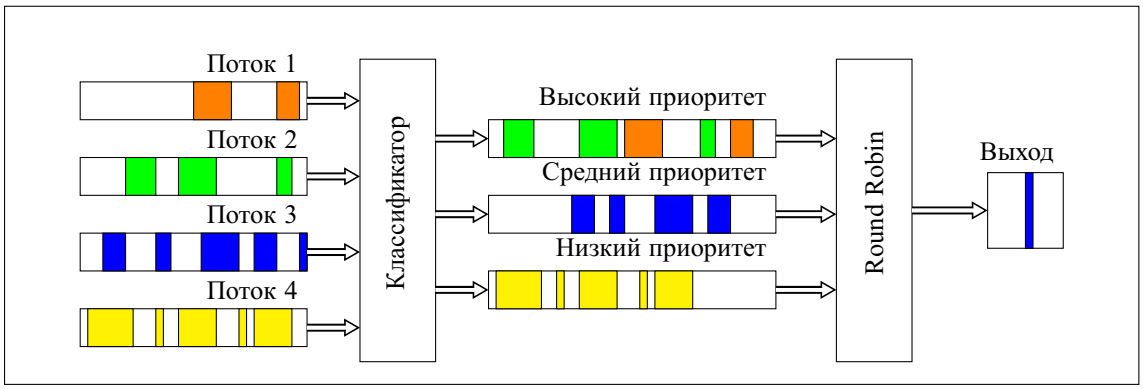


Рисунок 1 — Схема обслуживания алгоритмом приоритетных очередей

Приемущества алгоритма состоят в следующем:

– возможность понижения времени отклика, когда нет необходимости замедлять трафик [3];
– 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. [4]

Однако приоритетные очереди обладают рядом существенных недостатков.

– возникает проблема простоя канала (отсутствие обслуживания в течение продолжительного времени) для низкоприоритетного трафика при избытке высокоприоритетного[2];
– избыточный высокоприоритетный трафик может значительно увеличивать задержку и джиттер для менее приоритетного трафика;
– не решается проблема с TCP и UDP, когда TCP-трафику даётся высокий приоритет и он пытается поглотить всю пропускную способность. [4]

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. [5]

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.[2]

Алгоритм CBQ представляет канал в кажется иерархической структуры [6], пример которой представлен на Рисунке 2. Голубым цветом обозначен узел, представляющий собой основной канал; он разделяется между двумя классами трафика: интеркативным (левый узел) и остальным (правый узел), – которым назначается процент пропускной способности от остального канала. Весь трафик, причиляемый 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 [6].

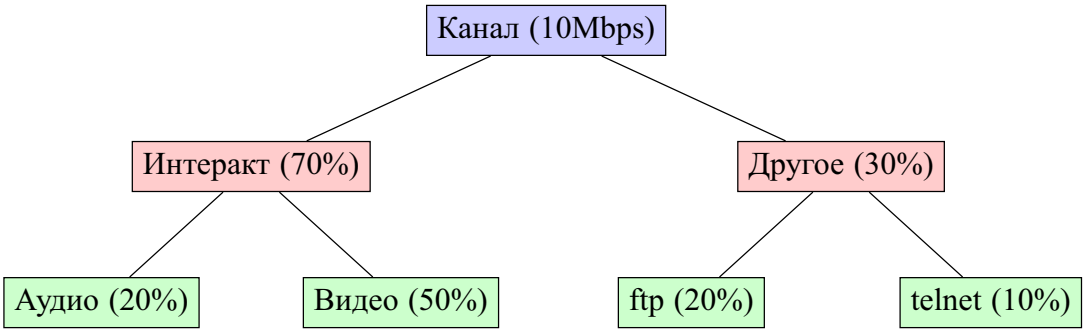


Рисунок 2 — Разделение канала при CBQ

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 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.[2]

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.[7]

Преимущества алгоритма состоят в следующем:

– позволяет контролировать количество пропускной способности для каждого класса обслуживания;

– каждый класс получает обслуживание, вне зависимости от других клас-

сов. Это помогает избегать проблемы PQ, когда при избытке высокоприоритезированного трафика низкоприоритезированный не обслуживался вообще.[2]

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 [2];
– высокая сложность реализации. В ядре Linux реализация CBQ приближённая и в некоторых случаях может давать неверные результаты.[7]

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 выделенно, оставшаяся пропускная способность распределяется между другими классами, которые требуют обслуживание.[8]

Отличительная особенность HTB от CBQ состоит в том, что в HTB принцип работы основывается на определении объема трафика[7], что даёт более точные результаты.

HTB состоит из произвольного числа иерархически организованных фильтров маркерного ведра (Token Bucket Filter, TBF)[2], однако реализация не использует готовый модуль `tbf`: алгоритм маркерного ядра втроен в код реализации HTB, что повышает его эффективность. Внутренние классы содержат фильтры, которые распределяют пакеты по очередям и метаинформацию, позволяющую функционировать механизм разделения канала. Листевые классы содержат очереди, которые содержат очереди, которые управляются сконфигурирован-

ными дисциплинами обслуживания (по умполчанию pfifo_fast).[Исходный код] Пример сконфигурированного дерева HTB представлен на Рисунке 3.

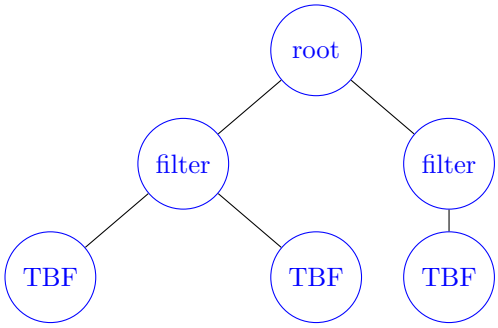


Рисунок 3 — Пример иерархии классов при использовании дисицплины HTB

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.[9] 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.[Исходный код]

Приемущества алгоритма HTB приведены ниже.

- Наиболее используемая дисциплина обслуживания в Linux, так как HTB эффективно справляется с обработкой пакетов, а конфигурация HTB легко масштабируется.[7]
- 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. [9]
- Вычислительно проще, чем алгоритм CBQ.[8]

Недостатки.

- Медленее CBQ в N раз, где N – глубина дерева разделения, что, однако, компенсируется простотой вычислений.[8]

– Нужно что-то ещё весомое.

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.[10]

HFSC ставит перед собой цели:

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

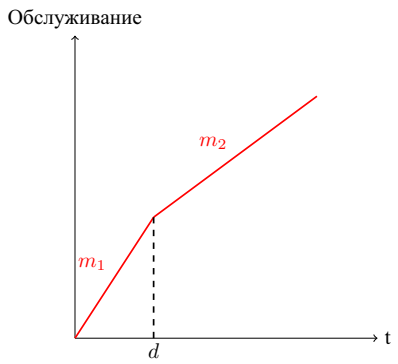
– честно выделять избыточную пропускную способность как указано классовой иерархией (критерий разделения канала);

– минимизировать несоответствие кривой обслуживания идеальной модели и действительного количество обслуживания.[11]

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.[11]

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.[12]

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



(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.[12]

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 листьевых классах, так как в этих вершинах непосредственно содержатся очереди.[10]

Основное преимущество алгоритма состоит в том, что он основан на формальной модели с доказанными нижними границами. Он даёт гаранитрованные 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 очередй на основе потоков

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 интер-

активный трафик в начало очереди, уменьшая там самым время ответа, и честно делит оставшуюся пропускную способность между остальными потоками. [13]

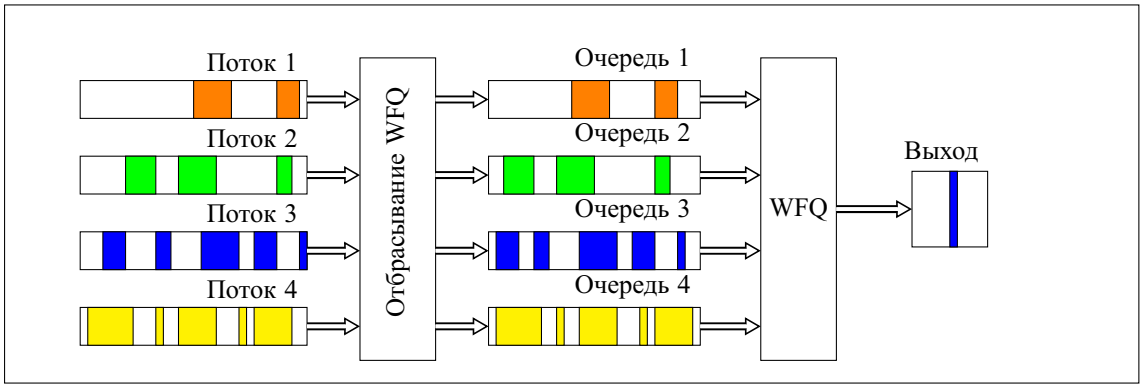


Рисунок 5 — Схема WFQ системы на основе потоков

Планировщик не нарушает порядка обработки пакетов, принадлежащих одному потока, даже в том случае, если они имеют различный приоритет. С этой целью поток реализуется в виде хэша, определяемого IP-адресом источника, IP-адресом цели назначения, полем протокола IP, номерами портов TCP/UDP и пятью битами байта ToS (Type of Service).Очередь потока обслуживается в соответствии с алгоритмом FIFO.

WFQ на основе потока использует для обработки каждого трафика так называемые очереди диалога (conversation queue). Поскольку память – конечный ресурс, число очередей диалога по умолчанию ограничено 256. Если число потоков превысит число очередей, допускается использование одной очереди для обработки нескольких потоков.[14]

В целях планировки в WFQ длина очереди измеряется не в пакетах, а во времени, которое заняла бы передача всех пакетов в очереди. WFQ адаптирует количество потоков и выделяет одинаковое количество полосы пропускания каждому потоку. Поток с маленькими пакетами, которые обычно являются интерактивными потоками, получают лучшее обслуживание, потому что они не нуждаются в большой полосе пропускания; также они получаются низкую задрежку, потому что у меньших пакетов меньшее время отправки (finish time). Время отправки – это сумма текущего времени и время, которое заняла бы отправка пакета. Текущее время ноль, если в очереди нет пакетов. WFQ поместит пакет в аппаратную очередь, основываясь на времени отправки в порядке возрастания.[И ОТКУДА ЭТО?]

Чтобы ввести вес в расчёт то, в каком порядке будут обслуживаться очереди, WFQ использует время окончания и приоритет IP (IP precedence). Вес расчитывается как время окончания, делённое на приоритет IP плюс один (во избежание деления на ноль). Однако для увеличения производительности в маршрутизаторах Cisco взамен времени отправки (finish time) используется размер пакета, так как он пропорционален времени; к тому же деление на приоритет IP заменяется на умножение фиксированного значения, просчитанного заранее (это сделано из-за того, что деление более трудная операция для CPU, чем умножение).[14]

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 достигнется 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 6. [15]

Приемущества WFQ.

- Простая конфигурация.
- Отбрасывание пакетов из более агрессивных потоков.
- Честное обслуживание.

WFQ страдает от нескольких недостатков.

- Трафик не может регулироваться на основе пользовательски определённых классов обслуживания.
- Не поддерживает задание определённой пропускной способности для типа трафика.
- В Cisco системах WFQ поддерживается только на медленных каналах.[16]

Эти ограничения были исправлены CBWFQ.

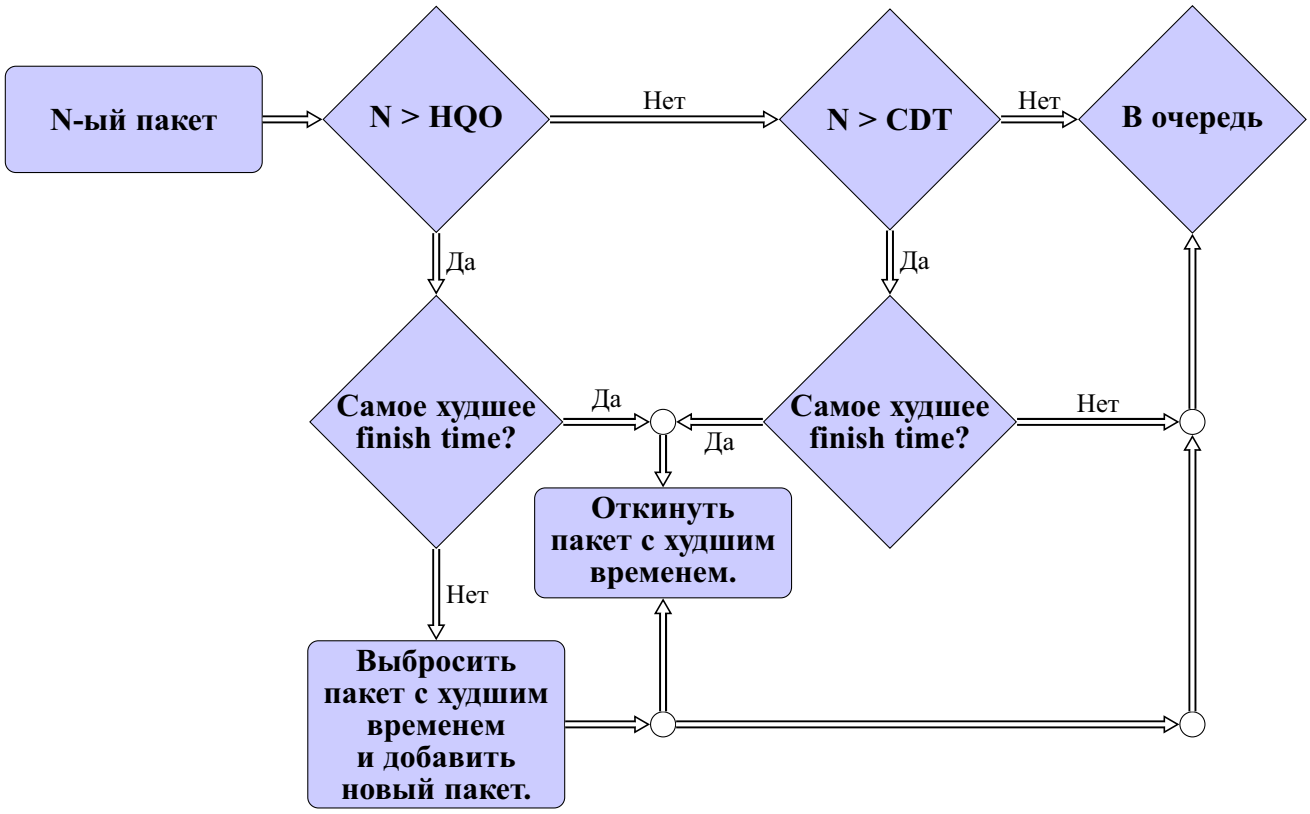


Рисунок 6 — Схема отбрасывания пакетов WFQ.

1.7 Взвешенный алгоритм честного обслуживания очередей на основе классов

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

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

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

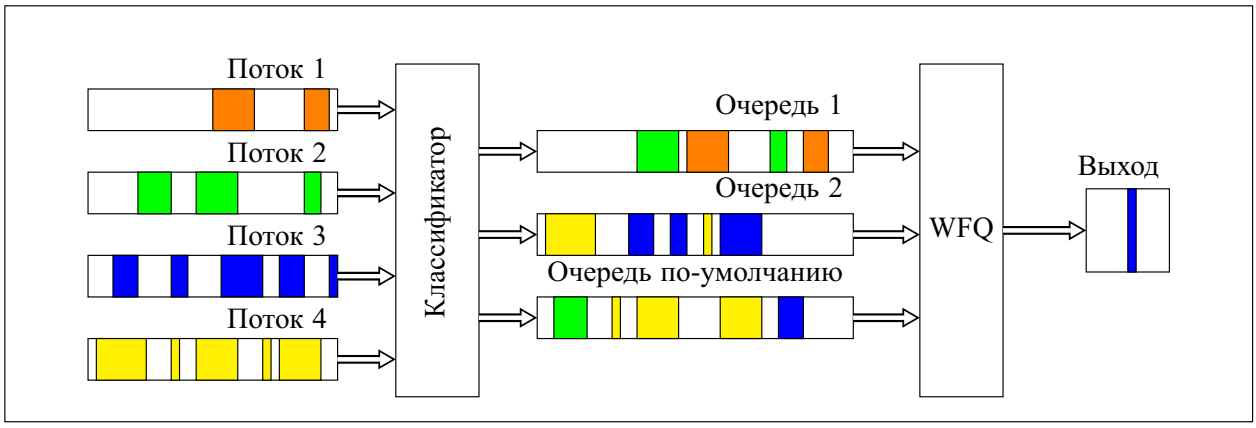


Рисунок 7 — Схема CBWFQ

задать в абсолютной (опция `bandwidth`), в процентной (опция `bandwidth percent`) и в доле от оставшейся полосы пропускания (опция `bandwidth 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). [13]

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.[13]

Преимущества:

- позволяет явно задать полосу пропускания для класса;
- позволяет содавать классы трафика и настраивать их в соответствии с требованиями;
- простая конфигурация вследствие небольшого числа параметров.[14][13]

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);[14]

– отсутствие открытой реализации, что усложняет реализацию алгоритма в других системах и требует его полного воссоздания на основе имеющихся источников.

1.8 Вывод

Каждая из рассмотренных ДО обладает своими достоинствами и недостатками. В Таблице 1 приведено сравнение основных элементов дисциплин обслуживания.

...

Поэтому реализация CBWFQ в ядре Linux целесообразна.

Свойство	**PQ**	**CBQ**	**HTB**	**HFSC**	**FWFQ**	**CBWFQ**
Метод планирования	RR	WRR	RR	RT/LS	WFQ	WFQ
Честность	-	-	-	+	+	+
Отбрасывание	TD	TD	TD	TD	ED/AD	TD/WRED
Разделение канала	-	+	+	+	-	-
Сложность реализации	H	B	C	B	C	C
Сложность конфигурации	H	B	C	B	H	H
Конфигурация классов	-	+	+	+	-	+
Реализация в Linux	+	+	+	+	-	-

Таблица 1 — Сравнительная таблица дисциплин обслуживания. Обозначения: RR – Round Robin (алгоритм циклического обслуживания), WRR – Weighted Round Robin (алгоритм взвешенного циклического обслуживания), RT/LS – Real-Time/Link-Sharing (алгоритм, который обслуживает очередь в зависимости от критерия реального времени и критерия разделения канала), TD – Tail Drop (алгоритм обрасывания "хвостов"), ED/AD – Early-Detection/Aggressive-Detection (алгоритм раннего и агрессивного обнаружения), WRED – Weighted Random Early Detection (алгоритм взвешенного раннего обнаружения), H – низная сложность, C – средняя сложность, B – высокая сложность (сложность оценивалась в относительно рассмотренных дисциплин).

2 РЕАЛИЗАЦИЯ CLASS-BASED WFQ В ЯДРЕ LINUX

2.1 Описание устройства подсистемы планировки в ядре Linux

В операционной системе Linux дисциплина обслуживания, обозначемая термином qdisc, используется для выбора пакетов из выходящей очереди для отправки на выходной интерфейс. Схема движения пакета приведена на Рисунке 8. Выходная очередь обозначена термином egress; именно на этом этапе следования пакета и работает механизм qdisc.[7]

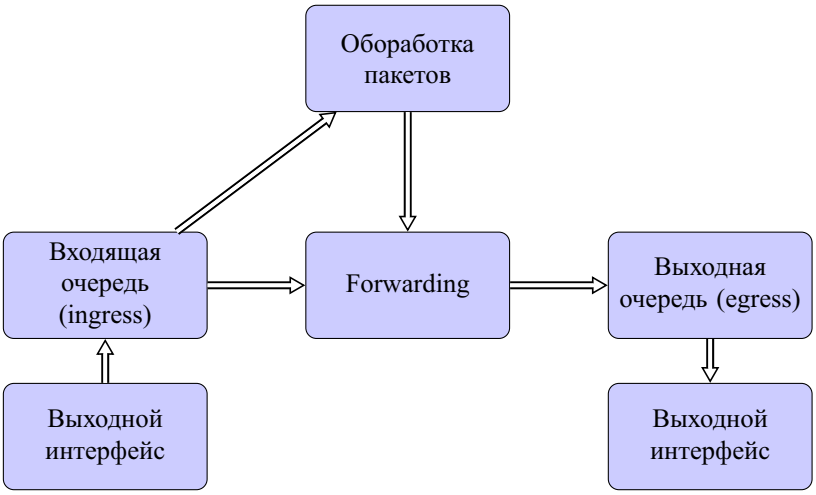


Рисунок 8 — Схема движения пакета в системе Linux[17]

В ощем случае, дисциплина обслуживания – это чёрный ящик, который может ставить пакеты в очереди и вынимать их из очереди, 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 дисцплины обслуживания. Примером бесклассовой дисциплины служит pfifo_fast, классовой – htb.[7]

Классы представляют собой одельные сущности в иерархии основной дисциплины. Если структура представляет собой дерево, то в классах-узлах могут содержаться фильтры, которые определят пакет в нужный класс-потомок. В классах-листьях непосредстенно располагаются очереди, которые управляются внутренней дисциплной обслуживания (обычно это FIFO).

Каждый интерфейс имеет корневую дисцплину (по умолчанию pfast_fifo), 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 9 представлен пример иерархии, основанной на описанных идентификаторов.

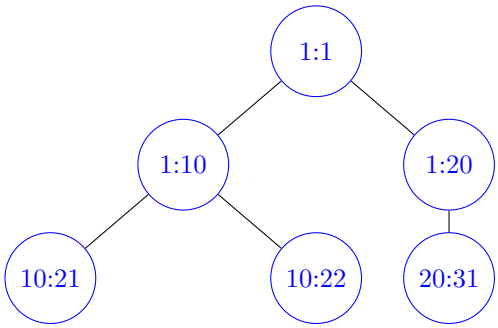


Рисунок 9 — Схема классовой иерархии с использованием идентификаторов MAJ:MIN

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

CBWFQ является классовой дисциплиной обслуживания с конфигурируемыми классами (в отличие от дисциплины PQ) и классом по умолчанию. Ядро предоставляет ряд полезных функций и структур данных, которые значительно упрощают написания необходимого кода в ядре и избавляет от совершения ряда критических ошибок. Таким образом, для наиболее эффективной реализации дисципилны обслуживания CBWFQ следует использовать предоставляемые ядром функции.

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 дисцплины 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.[17]

`tc` предоставляет интерфейс для дисцплины 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).

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 измнений и удаления используются соответственно `tc change` и `tc delete`.

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

Таком образом, для использования дисциплины обслуживания необходимо реализовать интерфейс в системе `tc`. Патчи для утилиты `tc` и ядра Linux, обеспечивающие взаимодействие между пользовательским пространством и дисциплиной представлен в Приложении [НОМЕР ПРИЛОЖЕНИЯ, ГДЕ ПАТЧИ ЛЕЖАТ].

2.3 Описание интерфейса

API ядра для подсистемы qdisc предосавтляет 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.[Исходный код]

Структура `struct Qdisc_ops` помимо метаинформации содержит указате-

ли на функции, которые должен реализовывать модуль дисциплины обслуживания для работы в ядре.[Исходный код]

Поля этой структуры представляют собой указатели на функции с сигнатурой, представленной ниже[Исходный код].

- `enqueue`

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

  Фукнкция добавляет пакет в очередь. Если пакет был отброшен, функция возращает 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 обслуживния, 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 другую структуру `struct Qdisc_class_ops`, которая описывает указатели функции исключительно для классовых дисциплин. Ниже приведены наиболее важные сигнатуры и их описания.[Исходный код]

- `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 иерархии классов.

- `tcf_block`, `bind_tcf`, `unbind_tcf`

  В данном случае, описание сигнатур не даст какой-либо значимой информации; 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 дисцплин, помимо описанного, реализуют классификацию пакетов, которая определяет класс, куда попадёт пакет. Классификация обычно выражается в функции `classify`, которая определяет, какому классу принадлежит пакет, и возвращает указатель на этот класс. Экземпляр структур для дисциплины обслуживания CBWFQ приведён в патче, представленном в Приложении [НОМЕР ПРИЛОЖЕНИЯ].

2.4 Алгоритм CBWFQ

2.4.1 Структура хранения данных Class-Based WFQ

Определене структуры представлено в Приложении [НОМЕР ПРИЛОЖЕНИЯ].

2.4.2 Добавление пакета в очередь

Описать enqueue, classify и drop. Все реализуют drop двумя способами: через qdisc_drop() и через if net_xmit_drop_count() { qdisc_qstat_drop(sch) }.

```
function ENQUEUE(Q, pkt)
    // Сначала нужно классифицировать пакет в очередь
    q ← CLASSIFY(Q, pkt)
    //Если количество пакетов в очереди не превосходит пороговых значений,
    //то мехнизм отбрасывания не запускается.
    //Иначе следует запустить соответствующий мехнизм отбрасывания.
    //Так как мехнизмы довольно схожи, их вычисление можно объединить.
    if q.count < HQO ∧ q.count < CDT then
        QDISC_ENQUEUE(q, pkt)
    else if pkt.size is the biggest then
        QDISC_DROP(pkt)
    else
        if q.count > HQO  then
            p_1 ← get_biggest_size(q)
            QDISC_DROP(p_1)
        end if
        QDISC_ENQUEUE(q, pkt)
    end if
end function
```

2.5 Тестирование модуля

Для тестирования модуля ядра была создана система виртуальных машин на основе системы эмуляции программного обеспечения QEMU. Схема тестовой среды представлена на Рисунке 10. Источником служит узел, от которого исходит трафик; траблицы маршрутизации настроены таким образом, чтобы весь

трафик, который должен попасть на узел-цель шёл через промежуточный узел, на котором настроена тестируемая дисциплина обслуживания CBWFQ.

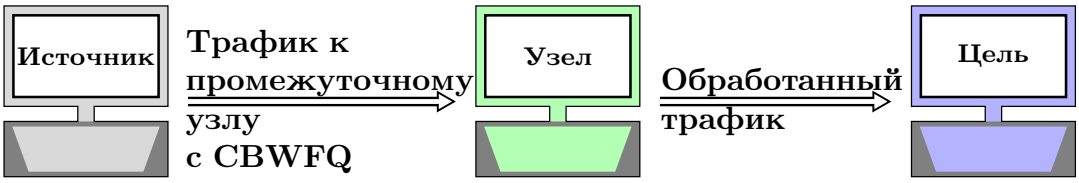


Рисунок 10 — Схема тестовой среды.

В Листинге 1 приведена система настройки дисциплины на промежуточном узле. Переменные окружения `TESTPORT1` и `TESTPORT2` содержат в себе номера портов, на которые направлен трафик (номера портов меняются). При добавлении дисципилны на интерфейс необходимо указать пропускную способность канала для корректных расчётов полосы пропускания для классов. При конфигурации классов во второй и третьей сторках происходит назначение каждому классу доли пропускной способности от канала в процентах (так же возможно указать точное количество пропускной способности в bps). В четвёртой и пятой строках происходит назначение фильтров, которые фактически определяют трафик для класса. На один класс можно назначить множество фильтров, посему это задача пользователя корректно определить трафик класса.

```
1 tc qdisc add dev eth1 cbwfq bandwidth 10
2 tc class add dev eth1 parent 1: classid 1:2 cbwfq rate 30 percent
3 tc class add dev eth1 parent 1: classid 1:3 cbwfq rate 60 percent
4 tc filter add dev eth1 parent 1: protocol ip u32 match ip sport
    $TESTPORT1 flowid 1:2
5 tc filter add dev eth1 parent 1: protocol ip u32 match ip sport
    $TESTPORT2 flowid 1:3
```

Листинг 1 — Список команд для конфигурации дисциплины обслуживания CBWFQ.

Для тестирования пропускной способности, которая была выделена каждому классу, была использована утилита iperf, которая запускалась на узле-источнике (Листинг 2) и узле-цели (Листинг 3). По умолчанию iperf генерирует

TCP-трафик с окном, указаным в командной строке на стороне сервера; с клиента на сервер передаётся трафик в три потока.

```
iperf -c $SERVERIP -P 3
```

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

```
iperf -s -w 1024
```

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

iperf выводит информацию о пропускной способности канала. Для указанной в Листинге 1 конфигурации после серии тестов были получены результаты, отображённые на Рисунке 11. TODO: сделать график получше.

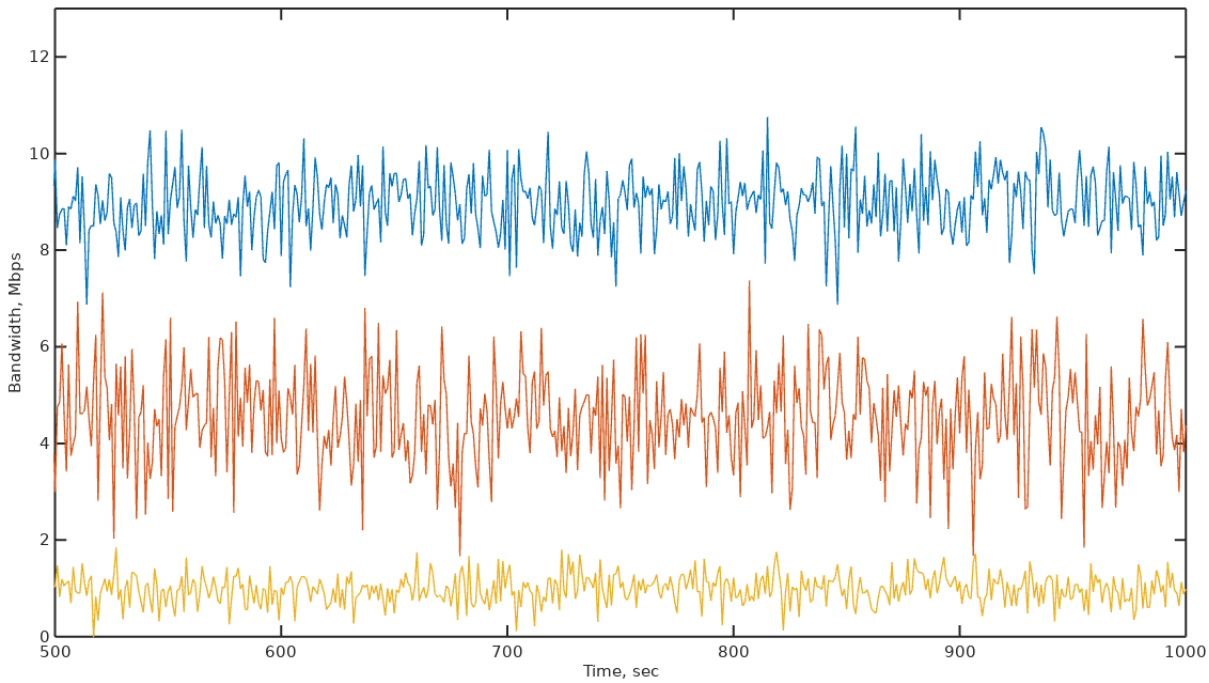


Рисунок 11 — График распределения пропускной способности по типам трафика в течение времени.

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Алиев Т.И. *Основы моделирования дискретных систем*. СПб: СПбГУ ИТМО, 2009.

[2] Davide Astuti. Packet handling. Department of Computer Science, University of Helsinki, 2003.

[3] Bert Hubert. man tc-prio, .

[4] Chuck Semeria. Supporting differentiated service classes.

[5] Alexey N. Kuznetsov. man tc-cbq.

[6] Floyd. Link-sharing blah-blah-blah.

[7] Bert Hubert. Linux advanced routing and traffic control, . URL: `lartc.org`.

[8] Martin devera, "htb linux queuing discipline manual - user guide". URL: `http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm`.

[9] Martin Devera. man tc-htb.

[10] T.S. Eugene Ng Ion Stoica, Hui Zhang. A hierarchical fair service curve algorithm for link-sharing, real-time and priority service.

[11] Michal Soltys. man tc-hfsc.

[12] URL: `http://linux-tc-notes.sourceforge.net/tc/doc/sch_hfsc.txt`.

[13] Cisco ios quality of service solutions configuration guide, release 12.2, 2009. URL: `https://www.cisco.com/c/en/us/td/docs/ios/qos/configuration/guide/12_2sr/qos_12_2sr_book.pdf`.

[14] Srinivas Vagesna. *IP Quality of Service*. Cisco Press, 2001.

[15] Quality of service, part 10 – weighted fair queuing, дата обращения 04.03.2018, . URL: `http://blog.globalknowledge.com/2010/02/12/quality-of-service-part-10-weighted-fair-queuing/`.

[16] Qos and queueing, aaron blachunas, . URL: `http://www.routeralley.com/guides/qos_queuing.pdf`.

[17] Some guy. *TCP/IP Architecture*. 1488.