Министерство образования и науки Российской Федерации
МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(государственный университет)
ФАКУЛЬТЕТ УПРАВЛЕНИЯ И ПРИКЛАДНОЙ МАТЕМАТИКИ
КАФЕДРА ТЕОРЕТИЧЕСКОЙ И ПРИКЛАДНОЙ ИНФОРМАТИКИ
(Специализация 010956
«Математические и информационные технологии»)

ЧЕСТНОЕ ПЛАНИРОВАНИЕ ДИСКОВОГО ВВОДА-ВЫВОДА С УЧЕТОМ РАЗДЕЛЯЕМЫХ РЕСУРСОВ ФАЙЛОВОГО КЭША

Магистерская диссертация
студента 873 группы
Храброва Алексея Викторовича

Научный руководитель
Костюшко А.В.

г. Долгопрудный
2014

Содержание

1. Введение

Одним из наиболее важных аспектов использования современных вычислительных систем является хранение данных. Системы хранения данных должны удовлетворять определенным требованиям надежности, гибкости и производительности.

Основными устройствами хранения цифровой информации являются жесткие диски (HDD – Hard Disk Drive). Работа с ними происходит медленно (на несколько порядков медленнее обращений к оперативной памяти), что в первую очередь связано с их механическим устройством: для перемещения считывающих головок к месту расположения нужных данных требуется значительное время (десятки миллисекунд). В последние годы получили распространение также SSD-диски (Solid State Drive), основанные на flash-памяти и не содержащие никакой механики. Они работают значительно быстрее HDD, но все же гораздо медленнее оперативной памяти. Поэтому для систем на их основе в целом характерны те же проблемы производительности, что и для систем с обычными жесткими дисками.

Дисковая подсистема зачастую становится наиболее узким местом многих систем, ограничивая скорость их работы. В частности, это характерно для высоконагруженных серверов, а также приложений мультимедиа. Оптимизация производительности дискового ввода-вывода – одна из важных задач, решаемых разработчиками операционных систем. В частности, было создано множество алгоритмов планирования дисковых операций.

Одна из наиболее важных проблем производительности состоит в обеспечении качества обслуживания для каждого отдельного клиента дисковой подсистемы – приложения, либо виртуальной машины или контейнера в системах виртуализации. Она решается т.н. честными дисковыми планировщиками, которые обеспечивают распределение пропускной способности между клиентами в соответствии с некоторыми заданными весами или приоритетами. В данной работе задача создания честного дискового планировщика решается для ОС Windows.

Существующие честные алгоритмы планирования не учитывают наличие неотъемлемого компонента дисковой подсистемы ОС – файлового кэша, поэтому даже при наличии честного распределения пропускной способности на уровне диска распределение на уровне приложений может быть нечестным. В данной работе проводится исследование этой проблемы и предлагается метод ее решения.

2. Цели и задачи

Целью данной работы является создание алгоритма планирования доступа к диску, обеспечивающего честность распределения пропускной способности дисковой подсистемы на уровне приложений, а не непосредственно на уровне диска, как существующие алгоритмы планирования. Предлагается строить алгоритм на основе одного из существующих подходов к планированию, в частности, реализованных в современных ОС, таких как Linux.

Для достижения данной цели требуется решить ряд вспомогательных задач.

1. Выбор существующего честного алгоритма планирования для использования на уровне диска. Алгоритм должен быть пригодным для добавления в него возможности учета использования файлового кэша. Далее будем называть этот алгоритм базовым.
2. Модификация выбранного базового алгоритма с целью корректного учета использования файлового кэша. Корректировка должна быть произведена таким образом, чтобы не ухудшить при этом общую производительность дисковой подсистемы. Полученный алгоритм должен будет обеспечивать честное распределение ресурса дисковой подсистемы над файловым кэшем.
3. Реализация построенного по данной схеме дискового планировщика в ОС Windows. Предполагается реализовать как оригинальный базовый алгоритм, так и модифицированный, с целью дальнейшего сравнения.
4. Проведение сравнительного анализа характеристик базового алгоритма и модифицированного при помощи тестов.

3. Обзор предметной области и ее проблематики

В этом разделе мы рассмотрим более подробно основные проблемы производительности дискового ввода-вывода, о которых говорилось во введении, а также обсудим существующие методы их решения.

3.1. Системы хранения данных и проблемы их производительности

С точки зрения задачи планирования ресурсов дисковый ввод-вывод во многом похож на другие типы ввода-вывода, например, передачу данных по сети, но в то же время имеет и некоторые принципиальные отличия. Главным из них является то, что в современных ОС диск используется не только как средство хранения пользовательских данных и программ, но и тесно связан с подсистемой управления памятью. В частности, диск, как устройство хранения, обладающее значительно большим объемом, чем оперативная память (но при этом значительно меньшей скоростью), используется ядром ОС для подкачки памяти (свопа) – сброса на диск неиспользуемых данных из оперативной памяти в случае ее нехватки. С подсистемой управления памятью также неразрывно связан файловый кэш – промежуточное хранилище данных в оперативной памяти с целью ускорения доступа к наиболее часто используемым данным.

Проблемы производительности подсистемы дискового ввода-вывода в современных вычислительных системах можно разделить на 2 основные категории, которые будут рассмотрены в следующих разделах:

- проблема обеспечения баланса между суммарной пропускной способностью диска и временем ожидания исполнения для отдельных запросов;
- проблема обеспечения качества обслуживания для каждого отдельного клиента дисковой подсистемы.

Далее мы рассмотрим некоторыми важные примеры проявления этих проблем в современных вычислительных системах.

Многим мультимедийным приложениям, в частности, проигрывателям музыки и видео, необходимо иметь возможность поддержания постоянной ширины канала на чтение данных с диска. При наличии в системе большого числа процессов, работающих с диском, многие системы оказываются неспособны обеспечить хоть и не широкий, но гарантированный канал для отдельных приложений.

Для высоконагруженных серверных систем (базы данных, web-серверы и т.п.) дисковая подсистема также является одним из узких мест. При высокой нагрузке малая скорость операций с файлами на диске может приводить к непозволительным задержкам в обработке клиентских запросов к серверу. Особенностью таких систем также является то, что помимо основных процессов, таких как сам сервер и системные процессы, в них присутствуют фоновые активности, такие как резервное копирование данных. Кроме того, одной из характерных для серверных систем проблема отказов в обслуживании при исчерпании ресурсов дискового ввода-вывода. В большинстве случаев клиенты при этом продолжают попытки подключения, что еще сильнее увеличивает нагрузку на сервер и может привести к еще большему количеству отказов.

В системах виртуализации проблемы производительности дискового ввода-вывода также являются актуальными, особенно проблема обеспечения качества обслуживания для отдельных виртуальных машин (или контейнеров в случае виртуализации уровня ОС). Основной задачей здесь является распределение ресурсов между виртуальными машинами в соответствии с заданными приоритетами. Важной задачей является также т.н. иерархическое планирование ресурсов, работающее на двух уровнях: распределение ресурсов между виртуальными машинами (или контейнерами) и распределение между процессами внутри виртуальной машины. Похожие проблемы возникают и в обычных (не виртуальных) многопользовательских средах. По аналогии с механизмом квот на дисковое пространство нужен такой же механизм для канала доступа к диску.

3.2. «Традиционные» дисковые планировщики

Проведем краткий обзор «традиционных» алгоритмов планирования доступа к диску, не ставящих своей целью решить задачу честности распределения ресурса дисковой подсистемы. Их можно разделить на 3 основные группы:

- алгоритмы, максимизирующие суммарную пропускную способность диска благодаря учету особенностей геометрии жестких дисков;
- алгоритмы, обеспечивающие ограниченное время ожидания исполнения отдельных запросов;
- алгоритмы, обеспечивающие некоторый баланс между суммарной пропускной способностью диска и временем ожидания для отдельных запросов.

В современных жестких дисках подобные алгоритмы, учитывающие относительное расположение запросов, реализуются в самих устройствах. Основным таким примером является технология SATA NCQ (Native Command Queueing), поддерживаемая в

большинстве современных систем хранения данных. Таким образом, учет геометрии жестких дисков для оптимизации суммарной пропускной способности перестает быть задачей, которую необходимо решать на уровне операционной системы.

Но в то же время учет относительного расположения запросов, в частности, т.н. локальности, на уровне ОС по-прежнему важен. Дело в том, что высокая степень локальности обращений к диску позволяет увеличить утилизацию аппаратного дискового кэша (количество попаданий запросов в кэш), что приводит к увеличению общей производительности диска.

Далее мы рассмотрим основных представителей каждого из данных классов алгоритмов дисковых планировщиков.

3.2.1. Планировщики для максимизации пропускной способности диска

Кратко опишем некоторые наиболее важные из алгоритмов, относящихся к данной категории.

- Алгоритмы SCAN (Elevator) и C-SCAN.

Приходящие от приложений запросы упорядочиваются в порядке их расположения в пространстве логических адресов на диске. При приходе нового запроса ввода-вывода в момент, когда диск простаивает, алгоритм выбирает в начальное направление обхода очереди запросов, совпадающее с направлением движения считывающей головки жесткого диска. Во время движения головки диска исполняются только запросы, расположенные в текущем направлении, в порядке их расположения. Когда головка доходит до конца диска, направление движения меняется на противоположное, и т.д. Таким образом, данный алгоритм напоминает работу лифта, благодаря чему и получил свое название.

В результате минимизируются времена передвижения головки к следующему запросу, и таким образом достигается увеличение общей пропускной способности диска. Однако время ожидания исполнения для отдельных запросов может быть слишком велико, из-за чего в явном виде данный алгоритм неприменим в системах, в которых стоит задача обеспечения качества обслуживания.

Существует также модификация данного алгоритма C-SCAN, отличающаяся тем, что считывающая головка все время движется в одном направлении, и при достижении конца диска перемещается в его начало. Обычно такой сброс головки выполняется значительно быстрее, чем ее «обычное» перемещение в обратном направлении во время исполнения запросов.

- Алгоритмы LOOK и C-LOOK.

В основе алгоритма LOOK лежит та же идея, что и в алгоритме SCAN. Отличие состоит в том, что при использовании алгоритма LOOK в случае, если после исполнения запроса в текущем направлении запросов больше нет, головка диска сразу начинает двигаться в противоположном направлении, не доходя до конца диска.

Аналогично SCAN, для алгоритма LOOK существует модификация C-LOOK, отличающаяся сбросом головки в начало диска вместо движения в противоположном направлении. Алгоритм C-LOOK используется для упорядочения запросов дискового ввода-вывода во многих современных ОС, в частности, в ОС Windows, а также в ОС Linux в качестве составной части планировщиков CFQ и BFQ, которые будут описаны в следующем разделе.

3.2.2. Планировщики для ограничения времени ожидания исполнения запроса

Алгоритмы, относящиеся к данной категории, также называют алгоритмами real-time планирования (режима реального времени). Для каждого запроса дискового ввода-вывода вводится т.н. deadline – момент времени, к которому запрос должен быть исполнен. Таким образом, при условии соблюдения deadline для каждого запроса планировщик обеспечивает гарантированное максимальное время ожидания исполнения для каждого запроса.

Простейшим алгоритмом такого рода является EDF (Earliest Deadline First), исполняющий запросы в порядке наступления их deadline. Существует также «гибридный» алгоритм SCAN-EDF, группирующий запросы согласно величине deadline: непрерывная ось времени делится на некоторые дискретные промежутки, и в одну группу попадают запросы, deadline которых лежит в одном и том же промежутке. Планировщик исполняет запросы по группам в порядке наступления deadline, причем в каждой группе запросы исполняются в порядке, определяемом алгоритмом SCAN.

Существует также класс приоритетных планировщиков, исполняющих запросы в порядке уменьшения приоритета, а внутри каждой группы по приоритету запросы исполняются в соответствии с одним из традиционных алгоритмов планирования. Примером приоритетного планировщика является планировщик, реализованный в ОС Windows начиная с версии Windows Vista/Windows Server 2008.

3.3. Проблема честного планирования доступа к диску и существующие решения

Перейдем к описанию класса т.н. честных алгоритмов планирования доступа к диску. Суть задачи честного планирования состоит в распределении ширины канала доступа к диску между многими процессами в соответствии с некоторыми заданными для этих процессов весами. При этом для каждого процесса гарантируется некоторая доля пропускной способности дисковой подсистемы в независимости от дисковой активности остальных приложений в системе.

Далее мы рассмотрим некоторые существующие честные алгоритмы планирования, причем особое внимание уделим алгоритму BFQ (Budget Fair Queueing), на основе которого в дальнейшем будет построен алгоритм для решения поставленной в данной работе задачи.

3.3.1. Простейшие алгоритмы

Большинство существующих на данный момент честных алгоритмов планирования доступа к диску в некотором смысле основаны на одном общем подходе – GPS (Generalized Processor Sharing). В оригинальной формулировке GPS предполагает, что распределяемый ресурс является непрерывным. Т.к. запросы ввода-вывода к диску являются дискретными и имеют в общем случае произвольный размер, необходимо построение некоторой аппроксимации GPS.

В качестве примеров простейших аппроксимаций GPS можно привести алгоритмы WRR (Weighted Round Robin), DRR (Deficit Round Robin) и WFQ (Weighted Fair Queueing). Их общая идея состоит в выделении каждому процессу очереди запросов и обходе этих очередей в некотором порядке. Во время обхода планировщик исполняет некоторое количества запросов из каждой из этих очередей в соответствии с логикой алгоритма, общей пропускной способностью и заданными весами процессов.

Рассматриваемые далее честные алгоритмы планирования являются развитием данных простых алгоритмов.

3.3.2. Алгоритм CFQ – Completely Fair Queueing

Планировщик CFQ является основным планировщиком в современных версиях ОС Linux и выставлен планировщиком по умолчанию во многих дистрибутивах этой ОС. Однако, в последние годы некоторые дистрибутивы, предназначенные для пользовательских систем, в частности, Ubuntu, начали отказываться от него в пользу более простых планировщиков, таких как deadline.

Алгоритм CFQ является развитием идеи Round Robin. Процессам с дисковой активностью выделяются некоторые кванты времени, в течение которых планировщик исполняет только запросы из очереди текущего процесса. Частота выдачи процессу кванта времени определяется его весом. В конце исполнения каждого кванта планировщик некоторое время ожидает прихода новых синхронных запросов, что решает т.н. проблему deceptive idleness («обманчивое бездействие»). Она заключается в том, что процесс с синхронной дисковой активностью не может отправить следующий запрос ввода-вывода, пока не будет исполнен предыдущий. С точки зрения алгоритма планирования такой процесс может выглядеть неактивным.

Благодаря предоставлению процессам эксклюзивного доступа к диску в течение коротких промежутков времени достигается высокая степень локальности обращений к диску, т.к. данные отдельно взятого процесса обычно располагаются на иске близко друг к другу. В результате достигается высокая суммарная производительность диска.

Один из основных и наиболее очевидных недостатков CFQ с точки зрения честности распределения ресурсов диска является следующая его особенность. Даже при честном распределении времени эксклюзивного доступа к диску распределение битрейтов может оказаться нечестным, т.к. пропускная способность диска меняется со временем при переходе планировщика от одного процесса к другому. Рассматриваемый далее алгоритм планирования BFQ данного недостатка не имеет.

3.3.3. Алгоритм BFQ – Budget Fair Queueing

Принципиальным отличием данного алгоритма от ранее рассмотренного CFQ является то, что он выделяет процессам не кванты времени, а т.н. бюджеты. Планировщик предоставляет текущему активному процессу эксклюзивный доступ к диску, пока процесс не исчерпает выделенный ему на данном шаге бюджет. Таким образом, планировщик на основе BFQ получает те же преимущества локальности обращений к диску, что и CFQ, но при этом может обеспечить более честное распределение битрейтов процессов в случае различной пропускной способности диска во время исполнения запросов от разных процессов.

Активный процесс выбирается на каждом шаге алгоритма на основании заданных весов процессов, а также уже полученного ими «сервиса» – количества байт, записанных на диск и прочитанных с диска. Для выбора активного процесса используется вспомогательный алгоритм WF^2Q+ (Worst-case Fair Weighted Fair Queueing). Изменения размеров бюджетов производятся исходя из текущей дисковой активности соответствующих процессов.

Приведем краткое описание вспомогательного алгоритма WF^2Q+, используемого для выбора активного приложения. Изначально данный алгоритм был разработан для планирования отправки пакетов в сетях передачи данных, и шаг алгоритма состоял в выборе следующего пакета для отправки. При использовании в рамках BFQ алгоритм WF^2Q+ на каждом шаге выбирает следующий активный процесс. Таким образом, роль пакета здесь играет совокупность запросов, исполняемых за период активности, суммарный размер которых равен бюджету процесса.

Алгоритм использует понятие виртуального времени (virtual time), измеряемого в байтах и соответствующего количеству полученного сервиса на данный момент реального времени. Системное виртуальное время равно суммарному количеству сервиса, полученного всеми приложениями в системе. Виртуальное время процесса равно количеству сервиса, деленному на вес приложения.

Приложениям, имеющим непустую очередь запросов к диску, присваиваются величины virtual start time, и virtual finish time, которые пересчитываются при вставке процесса в очередь ожидающих доступа к диску и при получении процессом некоторого количества сервиса. «Подходящими» называются процессы, для которых virtual start time меньше системного виртуального времени. В качестве активного выбирается тот процесс из «подходящих», который имеет наименьшее значение virtual finish time. В результате с течением времени процессы получают объемы сервиса, пропорциональные их весам.

3.4. Файловый кэш

В современных операционных системах для ускорения работы с более часто используемыми файлами использует файловый кэш – промежуточный буфер в оперативной памяти, содержащий фрагменты файлов с диска, которые могут быть запрошены пользовательскими процессами с наибольшей вероятностью. Доступ к данным в файловом кэше осуществляется на несколько порядков быстрее, чем к данным на диске, но его объем на несколько порядков меньше, причем он ограничен не только объемом оперативной памяти в системе, но и другими факторами, связанными с его реализацией в конкретных ОС. Очевидно, что наличие файлового кэша оказывает значительное влияние на производительность дисковой подсистемы, т.к. большая часть операций с файлами осуществляется приложениями именно кэшированным образом.

Наряду с кэшированием файлов в современных ОС существует механизм отображения файлов в память (memory mapped files). Процесс может работать с файлом, отобразив его в свое адресное пространство, при этом обращаясь к данным по

виртуальному адресу. При этом чтение данных из файла в память происходит посредством страничной ошибки (page fault) при первом обращении к ним, а запись является отложенной.

Кэширование файлов в современных ОС, в частности, в Windows и Linux, также реализовано на основе отображения файлов в память. Приведем краткое описание механизмов файлового кэша в ОС Windows. Для кэша выделены определенные диапазоны ядерных адресов, отображение файлов в которые описывается т.н. картой кэша (cache map) с гранулярностью 256 килобайт. Для каждого кэшированного фрагмента файла создается его отображение в соответствующий диапазон ядерной памяти, и чтение этого фрагмента с диска в кэш происходит путем вызова page fault-а. Помимо чтения данных в кэш непосредственно по запросу от приложения реализован механизм опережающего чтения (read ahead) – система заранее считывает с диска фрагменты файлов, которые с большой вероятностью будут запрошены приложением в ближайшее время.

Запись на диск измененных фрагментов кэшированных файлов происходит либо отложенным образом (write behind или write back), либо сразу после изменения (write through). Для механизма write behind в системе выделены специальные ядерные потоки, сканирующие файловый кэш в поисках измененных страниц и осуществляющие их сброс на диск. При этом в случае, например, неожиданного отключения питания возможна потеря пользовательских данных. Механизм write through же позволяет приложениям убедиться в том, что их данные действительно записаны на диск.

3.5. Аппаратный дисковый кэш

В современных жестких дисках и SSD-дисках применяется аппаратный дисковый кэш – промежуточный буфер между собственно жестким диском и использующей его операционной системой, физически расположенный на самом устройстве диска. В нем хранятся данные, которые были недавно считаны с диска или записаны на диск. Перечислим основные преимущества, даваемые наличием дискового кэша.

- Увеличение скорости чтения данных с диска за счет кэширования наиболее часто используемых данных и применения механизмов опережающего чтения.
- Ускорение записи за счет того, что контроллер диска сообщает ОС о завершении операции записи сразу же после ее прихода. Для обеспечения надежности при этом диском предоставляется возможность принудительного сброса данных и механизм write through.

- Совмещение скоростей передачи данных на физическом носителе информации и на интерфейсе ввода-вывода диска.
- Обеспечение возможности исполнения диском нескольких операций одновременно (с точки зрения ОС).

Наличие аппаратного дискового кэша оказывает значительное влияние на производительность работы системы с диском. Для увеличения утилизации дискового кэша планировщик дисковой активности должен учитывать его наличие. Иначе возможны ситуации, когда планировщик задерживает запросы ввода-вывода, которые могли бы попасть в кэш и быть исполнены значительно быстрее. Реализация такого учета осложняется тем фактом, что дисковый кэш является «прозрачным» для операционной системы – на уровне ОС нет никакой информации о том, присутствует ли конкретный диапазон данных в дисковом кэше. Это приводит к необходимости построения вероятностных моделей для получения такой информации на основании предыдущих операций ввода-вывода. В рамках данной работы данная задача не решается, однако она является естественным ее продолжением и остается в качестве одной из возможностей дальнейшего развития.

4. Проблема учета разделяемых ресурсов файлового кэша в дисковом планировщике

В этом разделе мы более подробно опишем проблему учета файлового кэша в алгоритмах планирования доступа к диску, рассматриваемую в данной работе. Мы обсудим два отдельных фактора этой проблемы:

- нечестность распределения самого ресурса файлового кэша;
- наличие в нем разделяемых между многими процессами данных.

Далее мы приведем результаты тестов, демонстрирующих проявление этих факторов, а также некоторые примеры проявления проблемы нечестности доступа к файловому кэшу в реальных системах. Кроме того, проведем обзор аналогичной проблемы, возникающей в задачах контроля за использованием физической памяти процессам и группами процессов в операционных системах, а также предлагаемых путей ее решения.

4.1. Проблема нечестного распределения ресурса файлового кэша

В современных операционных системах файловый кэш принято считать общим системным ресурсом. В частности, в ОС Windows, на которую в первую очередь ориентирована данная работа, файловый кэш «принадлежит» ядру системы в целом, и не ведется никакого учета его использования отдельными процессами. В результате возможна ситуация нечестного распределения ресурса файлового кэша между отдельными процессами в системе. Равноправные с точки зрения выделенных ресурсов физической памяти и пропускной способности диска процессы (или группы процессов) могут получать разные объемы файлового кэша.

Рассмотрим классическую (применяемую, в частности, в ОС Windows и ОС Linux) схему управления рабочим набором (working set) файлового кэша – набором страниц физической памяти, занимаемой кэшированными данными. Для вытеснения страниц используется алгоритм LRU (Last Recently Used) – ведется список страниц памяти, упорядоченный по времени последнего доступа к странице. В момент, когда нужно вытеснить страницу, выбирается та, которая наиболее давно использовалась. При этом список является общим на всю систему, в отличие от множества различных списков для рабочих наборов отдельных процессов.

Более того, ОС Windows файловый кэш является частью общего системного рабочего набора, в который входит также вся остальная физическая память, используемая ядром системы. Таким образом, в случае, например, возникновения паузы в использовании кэшированных файлов некоторым процессом (или группой процессов) соответствующие страницы из рабочего набора файлового кэша оказываются вытесненными, и в результате этот процесс получает меньшую долю ресурса файлового кэша.

4.2. Проблема учета разделяемых данных в файловом кэше

Перейдем к рассмотрению второго, не менее важного, фактора проявления нечестности в распределении ресурса файлового кэша между процессами. Рассмотрим следующую ситуацию.

Пусть группа процессов $\{P_1, \dots, P_n\}$ читают один из одного и того же общего (разделяемого) файла F, а также каждый процесс P_i читает из своего собственного файла F_i. Пусть заданные веса (приоритеты) в дисковом планировщике для всех процессов P_i равны. Тогда для честного распределения пропускной способности на уровне диска получаем, что битрейты (скорость передачи данных) для всех процессов P_i также равны между собой.

Пусть процесс P_1 (либо это могут быть несколько процессов из группы) систематически раньше читает разделяемый файл F, т.е. считывает фрагменты этого файла с диска в кэш чаще. Тогда остальные процессы будут читать эти данные преимущественно из кэша, и с точки зрения дискового планировщика ресурс диска при чтении этих данных не используют. В результате процессу P_1 будет выделено меньше ресурса на чтение своего собственного файла F_1, чем другим процессам P_i на чтение соответствующих файлов F_i.

Получаем, что общий битрейт на уровне файлового кэша у процесса P_1 меньше, чем у каждого из остальных P_i, даже при наличии честного распределения пропускной способности на уровне диска. Таким образом, наличие разделяемых данных в файловом кэше может приводить к нечестному распределению пропускной способности дисковой подсистемы на уровне приложений.

4.3. Примеры проявления в реальных системах

Приведем некоторые примеры реальных вычислительных систем, для которых наиболее заметно проявление изложенных ранее проблем. Все они отличаются

значительным количеством данных на диске, разделяемых между различными потребителями ресурсов дисковой подсистемы.

Основным примером являются системы контейнерной виртуализации (виртуализации уровня ОС). Запущенные на одной физической машине контейнеры используют значительное количество общих данных на диске. В частности, почти все системные файлы – исполняемые файлы ядра ОС, драйверов, системные библиотеки и т.п. – являются общими на все контейнеры в системе.

Другим примером являются системы с большим количеством виртуальных машин (или контейнеров), созданных из одного (базового) образа. Виртуальные машины при этом являются т.н. связанными клонами (linked clones), т.е. одинаковые для них данные хранятся на диске в единственном экземпляре. Одним из распространенных примеров таких систем является VDI (Virtual Desktop Infrastructure) – инфраструктура виртуальных рабочих столов. На серверах виртуализации в VDI запускается большое количество одинаковых виртуальных машин или контейнеров, клонируемых из базового образа, содержащего необходимый набор установленных приложений.

4.4. Обзор аналогичной проблемы учета разделяемой памяти

Рассмотрим аналогичную проблему, возникающую в задачах управления потреблением физической памяти процессами в операционных системах. Суть задачи состоит в контроле размеров рабочих наборов процессов и групп процессов и поддержании этих размеров в пределах заданных диапазонов. При этом важной задачей является определение принадлежности страниц физической памяти процессам в системе. При этом в современных системах значительная доля памяти является разделяемой между двумя и более процессами. В случае, если страница памяти является разделяемой, вопрос учета ее принадлежности при вычислении размеров рабочих наборов процессов становится нетривиальным.

Задача управления размерами рабочих наборов решается, в частности, контроллером памяти (memory controller) в подсистеме группового управления ресурсами cgroups в ядре ОС Linux, на основе которой строятся системы виртуализации уровня ОС (контейнерная виртуализация) для этой операционной системы. В текущей реализации cgroups страница памяти считается принадлежащей тому процессу (и соответствующей группе процессов), который первый начал ее использовать (первым вызвал page fault, в результате которого была выделена физическая страница). В целом данный подход работает достаточно хорошо, но при его использовании могут возникать систематические нарушения учета

принадлежности страницы, приводящие к нарушению честности распределения ресурса физической памяти в системе.

В частности, в случае, когда процесс, первым начавший использовать страницу памяти, завершается, эта страница продолжает быть принадлежащей группе, членом которой являлся данный процесс. При этом она может продолжать использоваться процессами из других групп, что контроллером памяти никак не учитывается. Другим примером является передача данных средствами IPC (inter-process communications) между процессами из разных групп по модели producer-consumer. Процесс-producer всегда обращается к разделяемым страницам памяти раньше, чем процесс-consumer, таким образом, именно producer будет считаться владельцем разделяемых страниц, хотя используются они в равной мере обоими процессами.

Для решения данной проблемы необходимо каким-либо образом записывать разделяемую страницу «на счет» всех процессов, которые ее используют. Одно из предлагаемых решений предполагает соответствующее расширение системного вызова *madvise()*, позволяющего процессу сообщить подсистеме управления памятью в ядре ОС дополнительную информацию об использовании им диапазонов памяти, в частности, разделяемой с другими процессами.

Для решения проблемы учета разделяемых данных в файловом кэше, рассматриваемой в данной работе, также необходимо получать информацию об использовании этих данных отдельными процессами. Однако в данном случае получение этой информации от самих процессов не является необходимым, т.к. обращения к файловому кэшу в любом случае обрабатываются ядром ОС, в отличие от обращений к физической памяти.

5. Создание алгоритма планирования для учета использования файлового кэша

Данный раздел посвящен основной задаче, решаемой в данной работе – построению алгоритма планирования доступа к диску, позволяющего обеспечить честное распределение пропускной способности дисковой подсистемы на уровне приложений, т.е. над файловым кэшем. Приводится математическая модель дисковой подсистемы и постановка основной задачи в рамках этой модели. Далее мы опишем предлагаемый метод решения задачи и приведем обоснование свойств полученного решения.

Предложенное решение состоит во внесении в существующий базовый алгоритм планирования необходимых модификаций для учета использования файлового кэша отдельными процессами в системе. При этом описанные в предыдущем разделе 2 фактора нечестности рассматриваются отдельно, и для каждого из них строится соответствующая модификация алгоритма. Приводится обоснование выбора BFQ в качестве базового алгоритма для модификации, а также анализ других возможностей построения дискового планировщика для учета файлового кэша.

5.1. Постановка задачи

Приведем формальные определения понятий, используемых при описании проблемы честного распределения пропускной способности диска, в частности, формализуем само понятие честности. Далее мы приведем математическую постановку задачи в терминах описанной модели дисковой подсистемы.

5.1.1. Математическая модель дисковой подсистемы

Для начала дадим формальные определения основных понятий, используемых при описании постановки задачи.

Запрос ввода-вывода на уровне приложений (над файловым кэшем):

$$R^C = (P, S, L^F, T_a, T_c, C) \quad (1)$$

где P – процесс-инициатор запроса,

S – размер запроса (в байтах),

$L^F = (F, O_F)$ – расположение запроса, совокупность файла и смещения в нем,

T_a – время прихода запроса, T_c – время его исполнения,

$C = (sync,\ cached,\ type)$ – класс, к которому относится запрос, в частности, является ли он синхронным или асинхронным, кэшированным или некэшированным.

Аналогично определяется запросы на уровне файловой системы (между файловым кэшем и файловой системой) и на уровне диска:

$$\mathrm{R^F} = (P, S, L^F, T_a, T_c, C), \qquad \mathrm{R^D} = (P, S, L^D, T_a, T_c, C) \tag{2}$$

с тем отличием, что расположение L^D есть смещение от начала диска. Отдельные параметры запросов будем обозначать как, например, $R^C.S$, либо $S(R^C)$.

Поток приходящих от каждого из приложений запросов представляет собой некоторый случайный процесс. Для моделирования этого потока может быть использован пуассоновский поток переменной интенсивности.

При прохождении через файловый кэш для запроса R^C порождается некоторое количество «подзапросов» $\{R_1^F,\ \dots,\ R_n^F\}$. Число подзапросов n может быть равным 0 в случае полного попадания запроса в кэш. Долей попадания в кэш (hit ratio) назовем следующую величину, равную 1 в случае полного попадания в кэш и равную 0 в случае полного промаха или некэшированного запроса:

$$h(\mathrm{R^C}) = 1 - \left(\sum_{i=1}^{n} S(R_i^F)\right) \Big/ S(\mathrm{R^C}) \tag{3}$$

Аналогично, при прохождении запроса R_F через файловую систему для него порождается совокупность подзапросов $\{R_1^D, \dots,\ R_n^D\}$, где $n \geq 1$, причем суммарный размер подзапросов $\sum_{i=1}^{n} S(R_i^D)$ может быть как равным $S(R^F)$, так и меньше его (в случае сжатых и разреженных файлов) или больше (в случае дополнительных обращений к метаданным файловой системы).

В отличие от оригинальной статьи с описанием алгоритма BFQ, мы не будем предполагать, что диск исполняет запросы один за другим, по одному за раз. Будем считать, что в каждый момент времени диск моет обрабатывать более 1 запроса. Исполнение запросов диском описывается функцией $T_c(T_a,\ S,\ L^D)$, которую можно рассматривать как некоторую случайную величину, определяемую параметрами диска и его текущим состоянием, в частности, состоянием дискового кэша.

Введем определение битрейта (скорости передачи данных), являющееся одним из центральных в постановке задачи. Итак, битрейтом процесса P будем называть следующую величину:

$$r_P(t) = \left(\sum_{R \in X} S(R)\right) / T, \qquad X = \{R \mid P(R) = P,\ T_c(R) \in [t - T, T]\} \tag{4}$$

Соответствующие функции определяются для битрейта на уровне приложений (над файловым кэшем) – $r_P^C(t)$, и на уровне диска – $r_P^D(t)$.

Величина промежутка времени T выбирается, исходя из следующих соображений. Она должна быть достаточно большой, чтобы битрейты процессов не были равны 0 большую часть времени (при наличии дисковой активности). В то же время T должно быть достаточно малым, чтобы функция $r_P(t)$ отражала изменения интенсивности дисковой активности процесса с течением времени.

Возможны и другие определения для функции битрейта, в том числе использующие различные методы сглаживания, а также учитывающие не только уже исполненные запросы, но и только отправленные на исполнение, однако в данной работе они рассматриваться не будут.

5.1.2. Определения характеристик дискового планировщика

Введем формальное определение понятия честности для планировщика дисковой активности. Пусть $S_i(t_1,\ t_2)$ – величина «сервиса», полученного процессом P_i за промежуток времени $[t_1,\ t_2]$, т.е.

$$S_i(t_1, t_2) = \sum_{R \in X} S(R), \qquad X = \{R \mid P(R) = P_i, T_c(R) \in [t_1, t_2]\} \qquad (5)$$

Алгоритм планирования будем называть честным, если он обеспечивает выполнение следующего условия:

$$\exists \delta: \forall t_1,\ t_2 \Rightarrow \left| S_i(t_1,\ t_2) - w_i \cdot \sum_{j=1}^{n} S_j(t_1,\ t_2) \right| \leq \delta, \forall P_i \in \{P_1, \dots, P_n\} \qquad (6)$$

где $\{P_1,\ \dots,\ P_n\}$ – множество процессов, имеющих непустую очередь запросов к диску в течение промежутка времени $[t_1,\ t_2]$. Авторами алгоритма BFQ было показано, что он является честным именно в терминах этого определения. Понятие честности вводится для дисковой активности процессов как на уровне диска, так и на уровне приложений (т.е. над файловым кэшем).

Приведем также альтернативное определение понятия честности. В нем используется т.н. метрика нечестности распределения битрейтов – величина, равная 0 в случае идеально честного распределения битрейтов, соответствующего весам процессов, и растущая с увеличением отклонения от этого идеального распределения. Приведем пример такой функции – сумма квадратов относительных отклонений битрейтов от битрейтов, соответствующих заданным весам:

$$f(t, w_1, \dots w_n) = \sqrt{\sum_{i=1}^{n} \left(\frac{r_i(t) - w_i \cdot \sum_{j=1}^{n} r_j(t)}{\sum_{j=1}^{n} r_j(t)} \right)^2} \quad (7)$$

где, как и в предыдущем определении честности, учитываются только процессы, имеющие дисковую активность (т.е. непустую очередь запросов к диску, ожидающих исполнения) в момент времени t.

В качестве альтернативного определения, алгоритм планирования будем называть честным, если он обеспечивает выполнение следующего условия:

$$\exists \sigma : \forall t \Rightarrow f(t) \leq \sigma \quad (8)$$

т.е. в некотором смысле минимизирует величину метрики нечестности в течение всего времени работы системы.

5.1.3. Цель корректировки алгоритма планирования

Сформулируем в терминах описанной ранее математической модели дисковой подсистемы те свойства дискового планировщика, которых мы хотим добиться путем корректировки базового алгоритма планирования с учетом использования процессами файлового кэша.

1. Минимизация нечестности распределения битрейтов процессов над файловым кэшем в терминах 2-го определения честности для случая нечестного распределения ресурса файлового кэша:

$$\left| S_i^C(t_1, t_2) - w_i \cdot \sum_{j=1}^{n} S_j^C(t_1, t_2) \right| \leq \delta \quad (9)$$

2. Обеспечение честности доступа к ресурсу дисковой подсистемы на уровне приложений в рамках 1-го определения честности для случая наличия разделяемых между несколькими процессами данных в файловом кэше:

$$f^C(t) = \sqrt{\sum_{i=1}^{n} \left(\frac{r_i^C(t) - w_i \cdot \sum_{j=1}^{n} r_j^C(t)}{\sum_{j=1}^{n} r_j^C(t)} \right)^2} \leq \alpha \quad (10)$$

3. Сохранение суммарной пропускной способности дисковой подсистемы:

$$\left(\sum_{i=1}^{n} r_i(t) \right)_{new} \geq \beta \cdot \left(\sum_{i=1}^{n} r_i(t) \right)_{original}, \quad \beta \leq 1, 1 - \beta \ll 1 \quad (11)$$

5.2. Анализ возможности расположения планировщика над файловым кэшем

Прежде чем приступить к описанию предложенного решения поставленной задачи, проанализируем возможность другого подхода к построению дискового планировщика для учета наличия файлового кэша, а именно расположения планировщика над файловым кэшем, а не непосредственно на уровне диска.

Все известные на данный момент честные алгоритмы планирования доступа к диску основываются на очередях запросов. Приходящие от приложений запросы к диску кладутся в соответствующие этим приложениям очереди, и далее в зависимости от алгоритма планирования исполняются в некотором порядке. Таким образом, начало исполнения для значительной части запросов к диску задерживается на некоторое время. Если расположить планировщик над файловым кэшем, он будет задерживать выполнение запросов, которые попадают в кэш и могли бы быть исполнены быстро. В результате может снизиться утилизация файлового кэша и общая производительность дисковой подсистемы.

Для обхода этого недостатка расположенный над файловым кэшем планировщик мог бы пропускать без задержки запросы, про которые известно, что они попадут в кэш. Но тогда такой планировщик будет фактически эквивалентен планировщику, расположенному на уровне диска.

Кроме того, расположенный над файловым кэшем планировщик не учитывает дополнительную дисковую активность, создаваемую файловой системой. К такой активности можно отнести, в частности, работу с метаданными файловой системы и запись в ее журнал. Возможен также и обратный эффект – например, при работе с т.н. разреженными (sparse) файлами и сжатыми файлами объем данных, прочитанных или записанных на уровне диска, меньше, чем на уровне приложений, работающих с такими файлами.

Таким образом, расположение планировщик дисковой активности на уровне файлового кэша является нецелесообразным. Необходимо создание планировщика, учитывающего запросы ввода-вывода, как на уровне диска, так и на уровне файлового кэша. В следующих разделах будет описано построение такого планировщика на основе корректировки базового алгоритма планирования на уровне диска с учетом информации об использовании процессами файлового кэша.

5.3. Динамическая корректировка весов процессов на уровне диска

Перейдем к рассмотрению общего метода корректировки базового алгоритма, учитывающего возможную нечестность распределения ресурсов файлового кэша между процессами. Суть его заключается в подстройке весов процессов в планировщике на уровне диска с целью уменьшения значения метрики нечестности для распределения битрейтов на уровне файлового кэша.

5.3.1. Описание метода корректировки

Выберем в качестве метрики нечестности распределения пропускной способности над файловым кэшем функцию, вычисляемую по формуле (7). Будем исходить из предположения, что битрейты в ней вычисляются на достаточно больших промежутках времени, чтобы было выполнено условие:

$$\forall P_i \;\Rightarrow\; \frac{r_i^D(t) \;-\; w_i^D \cdot R^D(t)}{r_i^D(t)} \ll 1, \qquad R^D(t) \;=\; \sum_{i\,=\,1}^{n} r_i^D(t) \tag{12}$$

Запишем выражение для метрики нечестности на уровне файлового кэша как функцию весов процессов w_i^D в базовом планировщике на уровне диска:

$$f(w_1^D, \ldots, w_n^D) \;=\; \sum_{i\,=\,1}^{n} \left(\frac{r_i^C \;+\; w_i^D \cdot R^D \;-\; w_i \cdot (R^C + R^D)}{w_i \cdot (R^C + R^D)} \right)^2 \tag{13}$$

Минимизируем данное выражение при условии равенства суммы этих весов единице, предполагая, что суммарная пропускная способность диска $R^D(t)$ изменяется с течением времени незначительно:

$$f(w_1^D, \ldots, w_n^D) \;\rightarrow\; min, \qquad \sum_{i\,=\,1}^{n} w_i^D \;=\; 1,\; w_i^D \;>\; 0 \tag{14}$$

Получаем задачу минимизации квадратичной функции при линейных ограничениях, решаемую аналитически методом множителей Лагранжа. Решение существует в силу выпуклости рассматриваемой функции и ограничений.

Корректировку весов предлагается проводить с некоторым «сглаживанием»:

$$(w_i^D)_{new} = (w_i^D)_{old} + \alpha \cdot \left((w_i^D)_{opt} - (w_i^D)_{old} \right) \tag{15}$$

где коэффициент α выбирается порядка $\frac{2}{3}$.

В модифицированном дисковом планировщике предполагается проводить корректировку весов по формуле (15) через некоторые промежутки времени. Среди возможных вариантов выбора величины этих промежутков можно отметить следующие:

- корректировка весов всех процессов через равные промежутки времени (порядка секунды);
- корректировка веса для отдельно взятого процесса при выборе его в качестве активного (специфично для алгоритмов наподобие BFQ).

5.3.2. Оценки алгоритмической сложности

Алгоритмическая сложность описанной корректировки весов процессов в планировщике на уровне диска зависит от того, каким именно образом и в какие моменты времени производить корректировку.

Если корректировка происходит для весов всех процессов через некоторые промежутки времени, то алгоритмическая сложность есть $O(n^2)$, т.к. при этом требуется для каждого из n процессов вычислить новое значение веса за время $O(n)$.

Если же корректировка происходит для отдельно взятого процесса, например, при выборе его в качестве активного в алгоритме планирования BFQ, то ее алгоритмическая сложность есть $O(n)$, где n – число процессов.

5.4. Обоснование выбора алгоритма на основе BFQ в качестве базового

Приведем обоснование выбора алгоритма планирования, используемого в качестве базового (на уровне диска) в данной работе – алгоритма на основе BFQ (Budget Fair Queueing), реализованного для ОС Linux.

В алгоритмах, основанных на выделении процессам бюджетов на выполнение запросов ввода-вывода, в частности в BFQ, все вычисления происходят в терминах прочитанных/записанных байт, в отличие от, например, CFQ – основного дискового планировщика в ОС Linux, который выделяет процессам некоторые кванты времени на исполнение запросов. Учет использования процессами файлового кэша также можно вести только в терминах размеров запросов. Поэтому алгоритмы на основе выделения бюджетов наилучшим образом подходят для корректировки для учета использования кэша.

BFQ является одним из наиболее эффективных среди существующих честных алгоритмов планирования доступа к диску, причем как с точки зрения честности распределения ресурса диска, так и с точки зрения суммарной производительности дисковой подсистемы. В частности, он демонстрирует более хорошие результаты в разнообразных тестах дисковой активности, чем CFQ. Кроме того, алгоритм BFQ обеспечивает гарантии честности распределения битрейтов процессов, а также

гарантированную верхнюю границу времени ожидания начала исполнения для каждого запроса.

Таким образом, представляется целесообразным выбрать BFQ в качестве основы для построения базового алгоритма дискового планировщика в данной работе.

5.5. Корректировка BFQ для учета разделяемых данных в файловом кэше

Перейдем к описанию метода корректировки базового алгоритма планирования, специфичного для алгоритма BFQ (Budget Fair Queueing), используемого в качестве базового. Суть метода состоит во внесении поправок в значения виртуальных времен процессов в алгоритме WF^2Q+ при чтении разделяемых данных процессами из файлового кэша. В результате использование процессами общих данных корректно учитывается планировщиком.

5.5.1. Описание метода корректировки

Суть метода состоит в том, что с точки зрения алгоритма планирования запрос на чтение с диска в кэш «делится» поровну между всеми процессами, которые впоследствии используют соответствующие данные, читая их из файлового кэша. Для этого вносятся соответствующие поправки в виртуальные времена процессов, используемые в алгоритме WF^2Q+ для выбора активного процесса.

При этом для процесса, прочитавшего фрагмент разделяемых данных с диска, виртуальное время уменьшается, для процессов, читающих этот фрагмент из кэша, увеличивается. Таким образом, процессы, читающие данные из кэша, получают за это некий «штраф», учитываемый планировщиком.

Пусть S – размер фрагмента данных, разделяемого между n процессами. P_D – процесс, прочитавший этот фрагмент с диска в файловый кэш. $\{P_A\}$ – множество, состоящее из $n \geq 1$ процессов, прочитавших фрагмент из кэша к настоящему моменту (в том числе процесс P_D). Тогда при чтении этого фрагмента из кэша процессом P_N поправки виртуального времени вычисляются следующим образом.

- Для каждого процесса P_i из множества $\{P_A\}$:

$$\Delta vtime_i = -\left(\frac{S}{n \cdot (n+1)}\right)/w_i \quad (16)$$

- Для процесса P_N:

$$\Delta vtime_N = +\left(\frac{S}{n+1}\right)/w_N \quad (17)$$

5.5.2. Свойства полученного алгоритма

В результате описанной коррекции виртуальных времен процессов в соответствии с формулами (16), (17) получаем, что в любой момент времени после завершения всех чтений из кэша некоторого фрагмента разделяемых данных верно следующее утверждение. Количество полученного сервиса (виртуальное время) для каждого процесса равно тому, которое этот процесс получил бы, если бы каждый запрос на чтение из диска в кэш разделяемых между n процессами данных размера S был заменен на n запросов размера S/n, отправленных каждым из этих процессов.

Таким образом, для временных промежутков длиной порядка характерного времени жизни данных в кэше (время от попадания в кэш до вытеснения из него) выполняется условие честности в терминах прочитанных/записанных байт за промежутки времени. Кроме того, такая корректировка виртуальных времен исключает возможность накопления систематической нечестности при работе с разделяемыми данными.

5.5.3. Оценки алгоритмической сложности

В следующей главе данной работы, при описании реализации систем мониторинга и планирования дискового ввода-вывода для ОС Windows, будет показано, что алгоритмическая сложность в среднем для операций со вспомогательными структурами данных, используемыми для хранения информации об операциях с разделяемыми данными в файловом кэше, есть $O(1)$.

Каждая операции корректировки виртуального времени отдельно взятого процесса происходит за время $O(1)$. Всего на каждую кэшированную операцию с разделяемыми данными выполняется $O(n)$ таких операций корректировки, где n – число процессов, обращающихся к разделяемым данным. Таким образом, алгоритмическая сложность корректировки есть $O(n)$ на каждую кэшированную операцию.

6. Реализация альтернативного дискового планировщика в ОС Windows

В данном разделе мы опишем практическую реализацию рассмотренных ранее алгоритмов в ОС Windows. Она включает в себя важную подзадачу – создание системы мониторинга дисковой активности, позволяющей отслеживать каждый запрос ввода-вывода на уровне диска и на уровне файловой системы. Система мониторинга позволяет определять принадлежность запросов конкретным процессам в системе, причем даже в сложных случаях, когда запрос исполняется в контексте ядра ОС. Также она позволяет вести учет использования процессами файлового кэша на уровне отдельных запросов и соответствующих фрагментов файлов.

Далее приводится описание архитектуры собственно планировщика дисковой активности и описание реализации базового алгоритма планирования (BFQ). Рассматриваются некоторые важные особенности реализации дискового планировщика, специфичные для ОС Windows.

Кроме того, приведены результаты сравнения рассмотренных алгоритмов планирования (оригинального BFQ и его модификации для учета файлового кэша) на тестах дисковой активности, демонстрирующие, что реализованный планировщик действительно решает поставленную задачу.

6.1. Система мониторинга дисковой активности

Постановку задачи мониторинга дисковой активности можно сформулировать следующим образом. Для каждого запроса дискового ввода-вывода, обрабатываемого системой, требуется определить процесс, являющийся реальным инициатором этого запроса. Сложность решения задачи состоит в том, что значительная часть файловых операций в ОС Windows обрабатываются в контексте ядра системы, а не того пользовательского процесса, который инициировал операцию. Примером такой операции ввода-вывода является сброс файлового кэша на диск.

Для решения этой задачи необходима глубокая интеграция в работу составляющих дисковой подсистемы в ядре ОС: драйвера файловой системы, менеджера файлового кэша, менеджера виртуальной памяти и драйвера диска. Реализация этого осложняется тем фактом, что Windows является системой с закрытым кодом. Это приводит к необходимости использования различных способов встраивания в систему, как

предусмотренных самой ОС, так и более сложных – требующих модификации бинарного кода системы во время исполнения.

Реализованный метод определения инициатора запроса в сложных сценариях состоит в том, что на начальной стадии исполнения запроса (когда процесс-инициатор еще известен) информация о запросе и его инициаторе (т.н. начальный контекст) сохраняется системой мониторинга. Затем, на основной стадии (при обработке запроса драйвером файловой системы или драйвером диска), производится поиск начального контекста по некоторым параметрам запроса, которые известны как на начальной стадии, так и на основной.

Помимо установления инициаторов отдельных запросов ввода-вывода, система мониторинга позволяет получать необходимую для корректировки алгоритма планирования информацию об использовании процессами файлового кэша. Далее мы рассмотрим архитектуру реализованной системы мониторинга и то, каким образом она обрабатывает исполнение запросов ввода-вывода в сценариях дисковой активности, связанных с использованием файлового кэша.

Для встраивания в процесс обработки запросов ввода-вывода используются 2 механизма – драйверы-фильтры и хуки функций ядра. Механизм драйверов-фильтров является штатным средством расширения функциональности подсистемы ввода-вывода и состоит в возможности встроить сторонний драйвер над (почти) любым из имеющихся в стеке драйверов. При этом фильтр перехватывает все запросы ввода-вывода, предназначенные целевому драйверу.

Механизм хуков функций заключается в модификации «на лету» (во время исполнения) бинарного кода ядра ОС таким образом, что при вызове целевой функции управление передается некоторому стороннему коду (этот код также называется хуком). Таким образом, при помощи установки хука можно «дописать» произвольный код в начало и конец любой функции ядра ОС.

6.1.1. Общая архитектура

Приведем описание предназначения каждого из компонентов системы мониторинга и реализованной в них функциональности. В следующем разделе на рисунке 3 изображена схема компонентов системы мониторинга, их расположение в дисковом стеке и взаимодействия с компонентами дисковой подсистемы ОС и с реализованным дисковым планировщиком.

Итак, система мониторинга дисковой активности состоит из следующего набора основных компонентов.

1. Драйвер-фильтр над файловой системой.

   Основой системы мониторинга является драйвер-фильтр, который встраивается в стек над устройствами смонтированных томов файловых систем. Этот фильтр перехватывает все IRP (I/O Request Packet, пакет запроса ввода-вывода) и Fast I/O (механизм обработки кэшированных запросов без создания IRP) запросы, как кэшированные, так и некэшированные, предназначенные драйверам файловых систем, а также встраивается в цепочку обработки завершения всех перехватываемых IRP.

2. Драйвер-фильтр под файловой системой.

   Используется для регистрации всего дискового I/O, в том числе инициируемого самой файловой системой. Данный фильтр расположен между драйвером файловой системы и драйвером диска и встраивается в стек устройств дисковых разделов.

3. Хуки функций менеджера файлового кэша и менеджера памяти.

   Используются для перехвата вызовов функций, связанных с кэшированным вводом-выводом и работой с отображаемыми в память файлами. Позволяют полноценным образом учитывать кэшированные операции, что особенно важно в контексте задачи обеспечения честного распределения битрейтов на уровне файлового кэша.

4. Контрольный драйвер.

   Центральный компонент системы мониторинга, осуществляющий управление и взаимодействие между остальными компонентами. В нем также реализован собственно планировщик ввода-вывода. Кроме того, он предоставляет интерфейс для приложений режима пользователя для забора статистики дискового ввода-вывода и выставления параметров алгоритма дискового планировщика, в частности, задания весов процессов.

6.1.2. Мониторинг кэшированного чтения и чтения memory mapped файлов

Обработка кэшированного чтения с точки зрения файловой системы разделяется на 2 стадии. Начальной стадией является приход первичного IRP или Fast I/O запроса, синхронного или асинхронного. Основной стадии соответствует обработка вторичного IRP чтения с диска. В случае синхронного запроса инициатором, очевидно, является текущий процесс. В случае асинхронного первичный IRP, перехватываемый драйвером-фильтром над файловой системой, связывается с последующим вызовом функции *CcCopyRead()* или *CcMdlRead()*, на которую устанавливается хук, через общие параметры: файл, смещение в файле, длина запроса.

Далее, при обработке драйвером вторичных IRP обработки страничных ошибок, чтение данных в кэш отсекается от других сценариев обработки page fault-ов на основании того, из какой функции этот page fault был вызван. В случае опережающего чтения инициатором считается последний процесс, инициировавший кэшированное чтение из соответствующего файла.

Таким образом, система мониторинга учитывает первичные кэшированные запросы от всех процессов, а не только от того, в контексте которого будет происходить чтение данных в кэш. Полученная информация об этих запросах передается в дисковый планировщик, который вносит необходимые корректировки.

Страничные ошибки, не связанные с файловым кэшем, можно разделить на 2 категории: page fault в страничном файле и page fault при чтении из memory mapped файла. Т.к. страничные ошибки обрабатываются синхронно, инициатором операции в данном случае является текущий процесс. Определение инициатора в случае page fault в странице, разделяемой между несколькими процессами, сильно усложняется. Проблема состоит в том, что в отличие от ядра ОС Linux, в ядре ОС Windows не используется механизм обратного отображения (reverse mapping), заключающийся в хранении сопоставления каждой странице физической памяти всех страниц виртуальной памяти, которые в нее отображаются.

Таким образом, не существует простого способа узнать список процессов, разделяющих данную страницу памяти или memory mapped файла. Тем не менее, можно получить список процессов, вообще имеющих доступ к разделяемому memory mapped файлу, и считать каждый из них инициатором операции чтения.

6.1.3. Мониторинг кэшированной записи и записи memory mapped файлов

Начальной стадией для записи в кэш, как и в случае кэшированного чтения, является обработка файловой системой первичного IRP или Fast I/O запроса кэшированной записи. Основной стадией является сброс диапазонов файла на диск из ядерного потока менеджера кэша функцией *CcFlushCache()*, на которую устанавливается хук. Сбрасываемый диапазон файла определяется из параметров внутренней функции менеджера памяти *MmFlushSection()*, на которую также устанавливается хук. Для первичных IRP сохраняется записываемый диапазон файла, по которому на основной стадии происходит поиск начального контекста. Аналогично устанавливаются инициаторы для операций явного сброса кэшированных файлов, которые также сводятся к вызову функции *CcFlushCache()*. В результате система мониторинга может определить

все процессы, писавшие в кэшированный файл, и информация о них передается в дисковый планировщик.

Перейдем к рассмотрению записи отображаемых в память файлов. Менеджер памяти ведет список модифицированных физических страниц, которые нужно сбросить на диск при нехватке оперативной памяти. Основной причиной попадания в этот список является вытеснение страницы из рабочего набора процесса. Вытеснение происходит в функциях *MiFreeWsle()* и *MiFreeWsleList(),* установка хуков на которые позволяет определить процесс, из рабочего набора которого страница была вытеснена.

На основной стадии (записи модифицированных страниц памяти на диск) происходит проверка, была ли каждая из записываемых на диск страниц ранее вытеснена, и по адресам этих страниц находятся все процессы, из рабочих наборов которых они были вытеснены. Эти процессы и считаются инициаторами операции. Более сложным сценарием является запись модифицированных страниц при уничтожении процесса. Определение инициатора в этом случае требует значительного вмешательства в структуры менеджера памяти.

6.1.4. Структуры данных для хранения информации об использовании кэша

Приведем описание структур данных, предназначенных для хранения информации об использовании процессами файлового кэша. Для сохранения информации о том, какие процессы считывают с диска в кэш определенные фрагменты файлов, используется т.н. «теневой кэш» (shadow cache).

Для каждого открытого на чтение файла в системе, с которым процессы работают кэшированным образом, хранится аналог карты кэша с гранулярностью от 64 до 256 килобайт в зависимости от размера файла. От величины этой гранулярности зависит итоговый расход системной памяти на теневой кэш. Было решено выбрать ее равной используемой менеджером кэша для хранения карты отображения кэшированных файлов в память, выделенную для кэша. В случае файлов меньшего размера, а таких в системе обычно большинство, хранится один элемент теневого кэша на весь файл.

Для каждого фрагмента файла в теневом кэше хранится процесс, который прочитал этот фрагмент, либо большую его часть, если таких процессов несколько, и список процессов, которые к настоящему моменту успели прочитать этот фрагмент или его часть из кэша. Кроме того, для файлов, открытых для записи, для каждого фрагмента хранится еще и список процессов, записавших в этот фрагмент.

Для организации поиска элемента теневого кэша по смещению в файле используются хеш-таблицы, роль ключа в которых выполняет смещение от начала файла, выровненное на величину гранулярности. Таким образом, операция поиска фрагмента кэша по смещению в файле выполняется в среднем за время $O(1)$. Операции вставки нового элемента и удаления существующего элемента также выполняются в среднем за время $O(1)$.

В то же время расход памяти на хранение теневого кэша является значительным. Потребление системной памяти на них всего на порядок меньше совокупных расходов менеджера кэша и менеджера памяти на хранение карт кэша и соответствующих отображений файлов в память.

Для удаления устаревших фрагментов теневого кэша используется перехват вытеснения соответствующих фрагментов из файлового кэша путем установки хука на функцию *CcFreeVirtualAddress()*, вызываемую при удалении соответствующего отображения (unmapping) файла в память. Таким образом, потребление памяти для хранения теневого кэша ограничено реально используемым объемом файлового кэша.

6.2. Особенности реализации в ОС Windows

Рассмотрим наиболее важные особенности реализации планировщика дискового ввода-вывода в ОС Windows. Эти особенности в основном связаны с отличиями подсистем дискового ввода-вывода в ОС Windows и в ОС Linux, для которой был изначально реализован алгоритм BFQ.

1. Большое количество асинхронных запросов (по сравнению с Linux). В частности, почти вся системная дисковая активность является асинхронной:
    - сброс файлового кэша и отображаемых в память (memory mapped) файлов;
    - опережающее чтение в файловый кэш (read ahead);
    - загрузка исполняемых образов и т.п.
2. Наличие запросов, исполнение которых нельзя задерживать надолго, иначе это может негативно отразиться на общей производительности системы:
    - сброс страниц физической памяти в файл подкачки в случае ее нехватки;
    - page fault-ы (ошибки отсутствия страниц), возникающие в коде ядра;
    - page fault-ы, возникающие в потоках, обрабатывающих очереди оконных сообщений и отрисовывающих графический интерфейс пользователя;
    - записи в журнал файловой системы NTFS с целью поддержки восстановления файловой системы в случае сбоя и т.п.

3. Наличие запросов больших размеров (вплоть до 8 мегабайт), исполнение которых занимает значительное с точки зрения планировщик время. Для сравнения, в ОС Linux размеры запросов дискового ввода-вывода ограничены 512 килобайтами.
4. Запросы ввода-вывода от пользовательских приложений могут разбиваться на более мелкие, например, драйвером файловой системы. В результате может возникать ситуация, когда дисковый планировщик исполнил только часть «подзапросов», и процесс вынужден ждать исполнения оставшихся «подзапросов» перед тем как отсылать следующие (в случае синхронной дисковой активности).
5. На том уровне стека драйверов дисковой подсистемы, куда удобно встраивать дисковый планировщик (в силу особенностей реализации драйверов уровня ядра в ОС Windows), реализована отличная от ОС Linux и оригинальной реализации BFQ модель взаимодействия с устройством диска. Основное отличие состоит в том, что в ОС Windows реализуемый нами дисковый планировщик (который является драйвером-фильтром в стеке драйверов диска) сам отсылает запросы ввода-вывода вниз к драйверу диска. В оригинальной реализации BFQ же предполагается, что драйвер диска сам запрашивает у планировщика следующий запрос на исполнение. В результате получается другая схема построения планировщика, описанная в следующем разделе.
6. Обработка завершения запросов ввода-вывода драйверами дискового стека в ОС Windows производится отложенным образом. При завершении операции ввода-вывода возникает прерывание от устройства диска, обработчик которого заказывает у системы DPC (deferred procedure call, отложенный вызов процедуры), которая будет исполнена при следующем вызове планировщика процессов. В этой DPC происходит обработка завершения запроса, и именно после ее исполнения процесс-инициатор запроса узнает о его завершении. Задержка между приходом прерывания о завершении запроса ввода-вывода и исполнением DPC может достигать величины периода системного таймера, значение которого по умолчанию составляет 16 миллисекунд. Данная схема делает еще более актуальным применяемое в алгоритме BFQ дополнительное ожидание синхронных запросов. В то же время, для возможности более точного измерения времени исполнения запроса необходимо либо уменьшать период системного таймера, что может привести к увеличению энергопотребления в системе, либо перехватывать первоначальные прерывания о завершениях запросов, что усложняет разработку планировщика.

6.3. Общая архитектура системы планирования

На рисунке 1 изображена общая схема архитектуры разработанного дискового планировщика. На схеме также приведены взаимодействия между компонентами планировщика, а также между планировщиком и составными частями подсистемы дискового ввода-вывода и компонентами системы мониторинга дисковой активности.

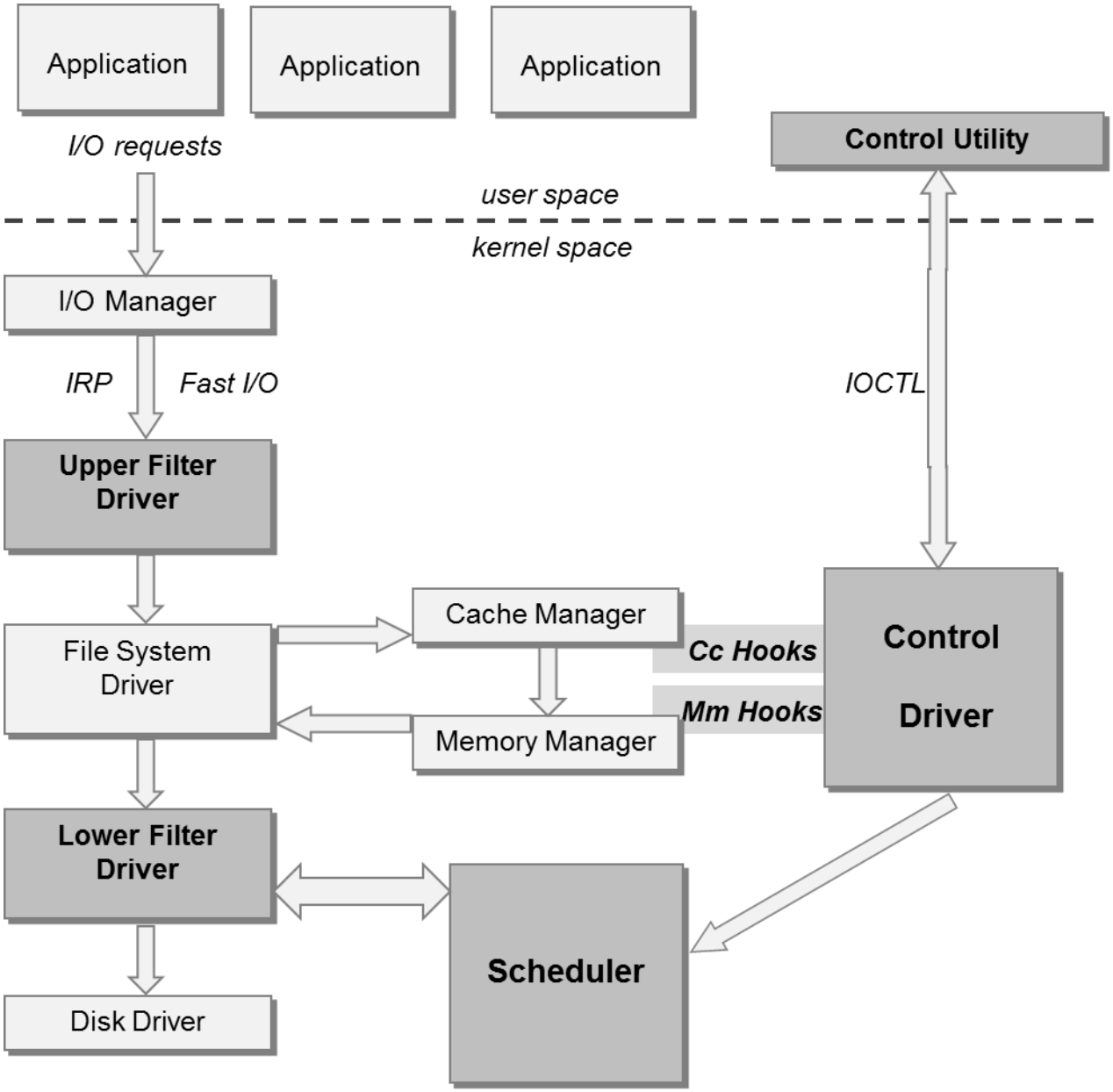

Рисунок 1

6.4. Реализация базового алгоритма планирования BFQ

Перейдем к описанию реализации дискового планировщика. Он расположен в контрольном драйвере и предоставляет интерфейсы для других компонентов планировщика и системы мониторинга. Схема работы планировщика и его взаимодействия с другими компонентами изображены на рисунке 2.

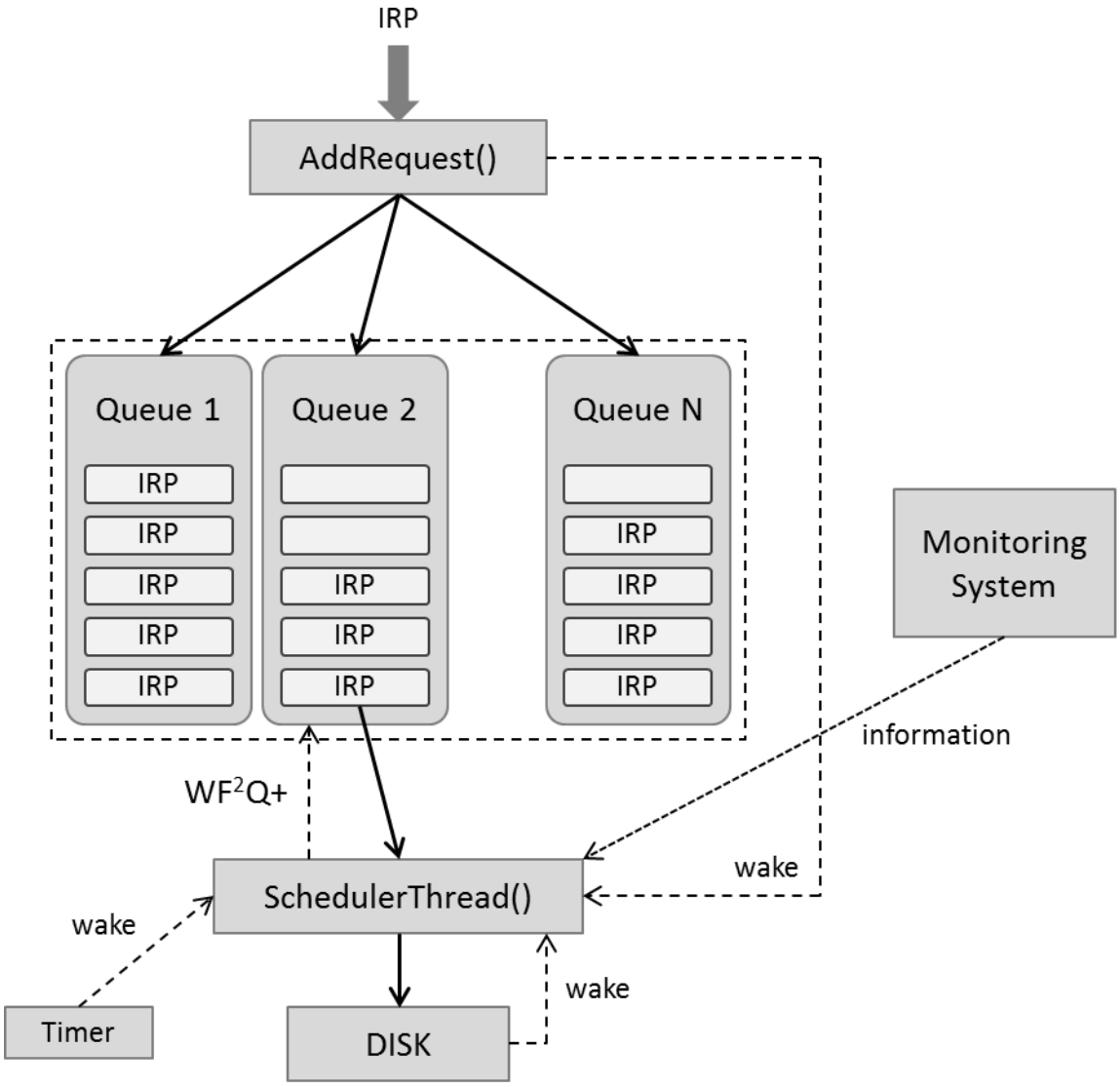

Рисунок 2

Нижний драйвер-фильтр (расположенный над драйвером диска) вызывает планировщик при приходе и при завершении каждого запроса ввода-вывода. При приходе запроса планировщик принимает решение, пропустить ли этот запрос вниз к драйверу диска сразу, на основании типа этого запроса, т.к. задержки некоторых запросов могут приводить к снижению общей производительности системы. Если же запрос не пропускается (это верно для большинства запросов), он поступает в распоряжение

планировщика, и нижний драйвер-фильтр переводит его в состояние pending (задержанный) и возвращает управление менеджеру ввода-вывода.

Далее соответствующий запросу пакет IRP помещается в одну из очередей запросов согласно тому, какой процесс является его инициатором. Имеется выделенная очередь на каждый процесс в системе (и, соответственно, на каждый диск, если в системе имеется несколько дисков).

На каждый диск в системе в планировщике создается отдельный поток исполнения, который пробуждается при возникновении любого из следующих событий:

- приход нового запроса ввода-вывода;
- завершение одного из ранее отправленных вниз запросов;
- по таймеру; период по умолчанию составляет 100 миллисекунд.

После этого, в соответствии с алгоритмом WF^2Q+, выбирается активный процесс, и поток планировщика начинает отправлять вниз запросы из очереди этого процесса.

Планировщик предоставляет функции для системы мониторинга, в которых пересчитываются виртуальные времена на основе использования разделяемых данных в кэше и подгоняются веса процессов на основе битрейтов над кэшем. Кроме того, планировщик предоставляет интерфейс для задания весов процессов.

Важным отличием от оригинальной реализации BFQ в ОС Linux является то, что в планировщике для ОС Windows нет необходимости реализовывать алгоритм C-LOOK, используемый для переупорядочивания запросов внутри очередей процессов с целью оптимизации их геометрии. Дело в том, что в дисковом стеке ОС Windows этот алгоритм уже реализован, причем в драйвере, расположенном ниже по стеку, чем реализуемый нами дисковый планировщик.

6.5. Сравнение рассмотренных алгоритмов планирования

Приведем результаты проведенных тестов, показывающие, что модифицированный планировщик дисковой активности на основе BFQ обеспечивает более честное распределение пропускной способности на уровне приложений по сравнению с оригинальным алгоритмом BFQ. Сравнение проводится между тремя планировщиками:

- NOOP, т.е. используется стандартный планировщик ОС,
- BFQ-original, реализация оригинального алгоритма BFQ,
- BFQ-cache-aware, реализация разработанного в ходе данной работе алгоритма, учитывающего наличие файлового кэша.

В ходе тестов некоторое число процессов читают общие данные с диска, причем некоторые из них считывают фрагменты файлов раньше, чем другие. Таким образом, тесты пока имеют синтетический характер. Кроме того, они в значительно большей степени отражают влияние фактора разделяемых данных, чем фактора нечестности распределения ресурса файлового кэша. Для проведения тестов реальных систем, в которых проявляется проблема нечестности использования кэша, в частности, систем виртуализации, необходима интеграция планировщика в эти системы виртуализации, что представляет собой отдельную инженерную задачу.

6.5.1. Честность распределения пропускной способности над файловым кэшем

На рисунке 3 изображен график сравнения показателей нечестности распределения пропускной способности на уровне приложений (квадратичное относительное отклонение от идеального распределения) для рассмотренных алгоритмов при различном числе читающих разделяемые данные процессов.

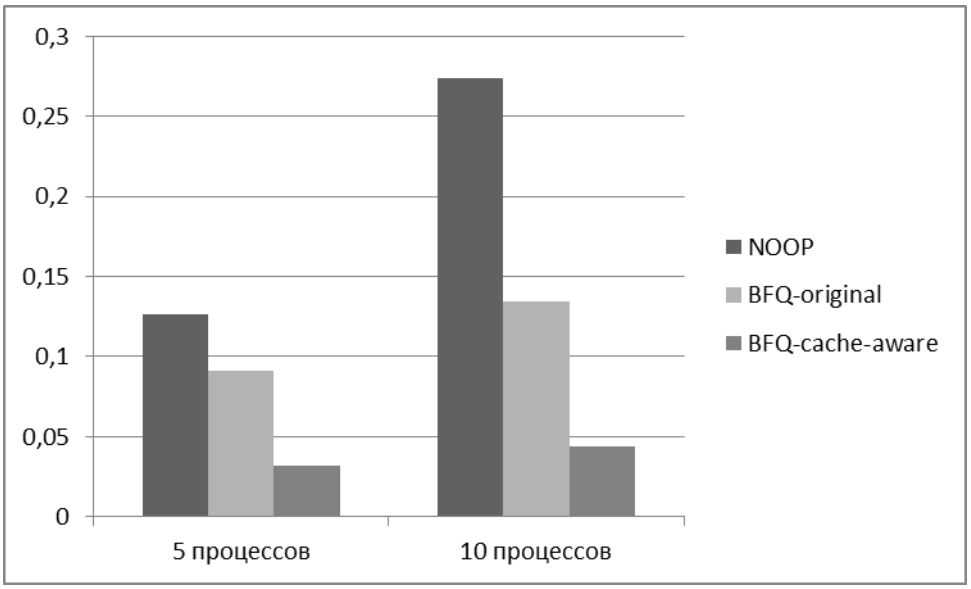

Рисунок 3

Из графиков видно, что модифицированный алгоритм обеспечивает более честное распределение битрейтов на уровне приложений.

6.5.2. Общая производительность дисковой подсистемы

На рисунке 4 изображен график сравнения суммарной пропускной способности дисковой подсистемы (совокупная пропускная способность диска и файлового кэша, мегабайт в секунду) для рассмотренных алгоритмов при различном числе читающих разделяемые данные процессов.

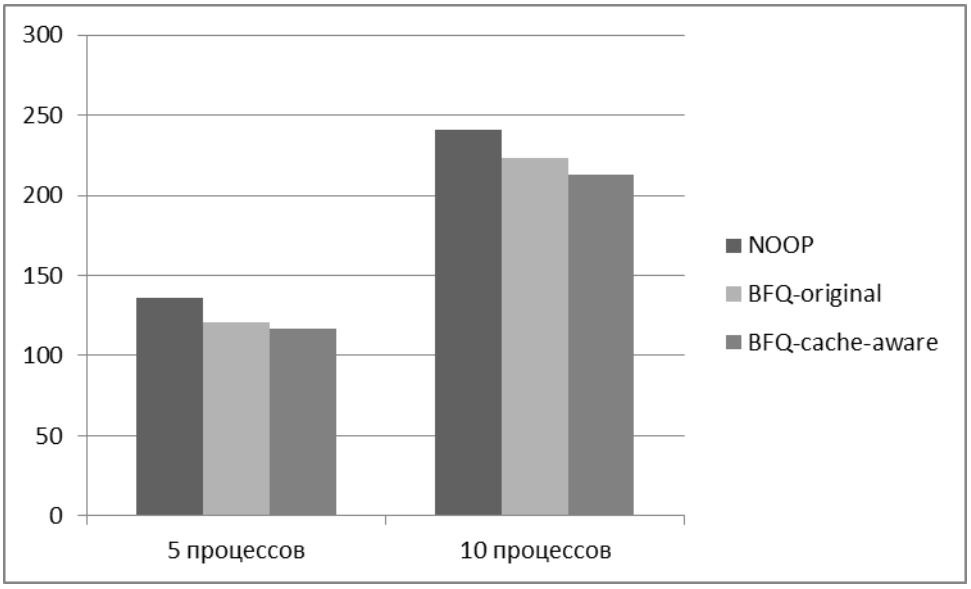

Рисунок 4

Из графиков видно, что падение общей производительности дисковой подсистемы при использовании модифицированного алгоритма является незначительным.

7. Заключение. Результаты и пути дальнейшего развития

Целью данной работы являлось создание алгоритмы планирования дисковой активности, обеспечивающего честность распределения ресурсов дискового ввода-вывода на уровне приложений, а не только на уровне диска.

В рамках работы было проведено исследование существующих алгоритмов честных дисковых планировщиков. Был сделан вывод о том, что существующие решения не учитывают наличие разделяемого между различными процессами ресурса дисковой подсистемы – файлового кэша. На основании этого была поставлена задача построения честного дискового планировщика, который учитывает использование процессами этого разделяемого ресурса.

Для решения поставленной задачи было предложено взять уже существующий алгоритм планирования для использования в качестве базового (на уровне диска) и внести в него некоторые модификации для корректного учета кэшированной дисковой активности. В качестве базового алгоритма был выбран BFQ, и был разработан способ его корректировки для обеспечения честности на уровне файлового кэша. Был реализован прототип разработанного алгоритма для операционной системы Windows.

Среди возможных путей дальнейшего развития проекта можно отметить следующие.

- Тестирование реализованного прототипа на реальных системах – серверах с большим количеством контейнеров или виртуальных машин с общими данными.
- Доработка предложенного алгоритма подстройки весов процессов в планировщике на уровне диска с целью минимизации нечестности на уровне кэша, в частности, поиск других возможных метрик для нечестности распределения пропускной способности дисковой подсистемы.
- Учет аппаратного дискового кэша в алгоритме планировщика; построение необходимой для этого в силу прозрачности дискового кэша для операционной системы его вероятностной модели.
- Учет внутреннего дискового планировщика в алгоритме планирования.

Список использованных источников

1. *P. Valente, F. Checconi*. High throughput disk scheduling with fair bandwidth distribution. // IEEE Transactions on Computers, 2010.
2. *P. Valente*. New version of BFQ, benchmark suite and experimental results. [Электронный ресурс] http://algo.ing.unimo.it/people/paolo/disk_sched/bf1-v1-suite-results.pdf
3. *P. Valente, M. Andreolini*. Improving Application Responsiveness with the BFQ Disk I/O Scheduler. // Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR '12). ACM, 2012.
4. *J.C.R. Bennett, H. Zhang*. Hierarchical packet fair queueing algorithms. // IEEE/ACM Transactions on Networking, 1997.
5. *B.L. Worthington, G.R. Ganger, Y.N. Patt*. Scheduling algorithms for modern disk drives. // SIGMETRICS '94, 1994.
6. *S. Iyer, P. Druschel*. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. // 18th ACM SOSP, 2001.
7. *A.L.N. Reddy, J. Wyllie*. Disk scheduling in a multimedia I/O system. // MULTIMEDIA '93: Proceedings of the first ACM international conference on Multimedia. ACM, 1993.
8. *A.K. Parekh, R.G. Gallager*. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. // IEEE/ACM Transactions on Networking, 1993.
9. *M. Shreedhar, G. Varghese*. Efficient Fair Queueing Using Deficit Round Robin. // IEEE/ACM Transactions on Networking, 1996.
10. *H.P. Chang, R.I. Chang, W.K. Shih, R.C. Chang*. Real-Time Disk Scheduling with on-Disk Cache Conscious. Lecture Notes in Computer Science, 2003.
11. *D. Solomon, M. Russinovich*. Windows Internals, 6th edition. Microsoft Press, 2012.
12. *D. Probert*. Lectures on Windows Kernel at the University of Tokyo. [Электронный ресурс] http://i-web.i.u-tokyo.ac.jp/edu/training/ss/lecture/new-documents/Lectures
13. Microsoft. Windows Research Kernel. [Электронный ресурс] http://www.microsoft.com/education/facultyconnection/articles/articledetails.aspx?cid=2416
14. *Love R.* Linux Kernel Development. 3rd edition. Addison-Wesley, 2010.
15. Linux Kernel Documentation. CFQ (Complete Fairness Queueing). [Электронный ресурс] https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt

16. Linux Kernel Documentation. Block I/O Controller. [Электронный ресурс] https://www.kernel.org/doc/Documentation/cgroups/blkio-controller.txt
17. *G. Thelen, M. Lespinasse*. Shared-memory accounting in memory cgroups [Электронный ресурс] https://lwn.net/Articles/516541
18. Linux Kernel Documentation. Memory Resource Controller. [Электронный ресурс] https://www.kernel.org/doc/Documentation/cgroups/memory.txt
19. *Храбров А.В., Рубанова Ю.В*. Мониторинг дисковой активности в ОС Windows. // Труды 54-й научной конференции МФТИ «Проблемы фундаментальных и прикладных естественных и технических наук в современном информационном обществе», 2011.
20. *Храбров А.В., Рубанова Ю.В*. Планирование ресурсов дискового ввода-вывода в ОС Windows. // Труды 55-й научной конференции МФТИ «Проблемы фундаментальных и прикладных естественных и технических наук в современном информационном обществе», 2012.