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»)

РАЗРАБОТКА ЧЕСТНОГО АЛГОРИТМА ДИСКОВОГО ПЛАНИРОВЩИКА ДЛЯ ОС WINDOWS НА ОСНОВЕ ВЫДЕЛЕНИЯ БЮДЖЕТОВ С ОБЕСПЕЧЕНИЕМ КАЧЕСТВА ОБСЛУЖИВАНИЯ НА КОРОТКИХ ВРЕМЕННЫХ ИНТЕРВАЛАХ

Магистерская диссертация
студентки 873 группы
Рубановой Юлии Викторовны

Научный руководитель
Костюшко А.В.

г. Долгопрудный
2014

Оглавление

1. Введение

Диск – самый медленный вид памяти современных компьютеров. Зачастую быстродействие приложений определяется именно скоростью взаимодействия системы с диском. С развитием интернета проблема низкой производительности дисковой подсистемы стала особенно актуальной, т.к. веб-серверу нужно постоянно читать с диска большие объемы данных, в то время как пользователи, запрашивающие данные у веб-сервера, не готовы ждать продолжительное время.

Предметом внимания в данной работе является планирование ввода-вывода в дисковой подсистеме ОС Windows. До сих пор для данной системы не создано программ, способных эффективно оптимизировать запросы ввода-вывода. Это связано, в первую очередь, со сложной структурой этой ОС, а также с закрытостью ее кода. Первой задачей данной работы является реализация одного из существующих алгоритмов планирования для ОС Windows.

Существует множество алгоритмов планирования ввода-вывода. Лучшие из них с точки зрения обеспечения честности [5], такие как BFQ [1] и CFQ [6], реализованы под ОС Linux и широко применяются для оптимизации дисковой активности на серверных системах. Однако они обладают существенными недостатками. Будучи разработанными для серверных систем, эти алгоритмы не учитывают, комфортно ли пользователю будет работать на машине с таким планировщиком.

Вторая и наиболее важная цель данной работы – разработать планировщик дисковой активности в ОС Windows, подходящий для работы на машине пользователя.

2. Основные понятия

Введем формальные понятия, которыми будем оперировать в представленной работе.

Когда какое-либо приложение желает считать или записать что-то на диск, оно формирует *запрос* (в случае Windows это IRP пакет), посылаемый в файловую систему, которая в свою очередь пересылает его к диску, если запрашиваемой страницы нет в кэше.

Приложения могут отправлять запросы ввода-вывода двух основных типов: синхронные и асинхронные.

- **Синхронный ввод-вывод.**

При этом типе запроса приложение ждет, пока запрос не будет выполнен и не вернул код статуса завершения операции ввода-вывода. После такой операции полученные данные могут быть немедленно использованы. Большинство приложений использует запросы именно такого типа.

- **Асинхронный ввод-вывод**

Позволяет приложению отправить запрос на ввод-вывод, но не дожидаться получения данных с устройства. Это позволяет решать другие задачи, пока выполняется операция ввода-вывода. Очевидным минусом является невозможность использовать данные с диска немедленно. Также при этом нужно следить за тем, чтобы не было обращений к данным до их получения с диска.

Битрейтом процесса называется скорость, с которой процесс читает или пишет данные на диск. Механизм жесткого диска устроен так, что время чтения или записи одного и того же количества байт может сильно меняться в зависимости от текущего местонахождения головки диска, от того, шлет ли процесс запросы с последовательным расположением на диске или нет, от того, насколько часто диск переключается на исполнение запросов от другого процесса, и других факторов.

Чтобы увеличить суммарный битрейт и битрейт каждого процесса в отдельности, применяются планировщики дисковой активности. В алгоритмах планирования каждому процессу сопоставлено некоторое значение, называемое *весом*. При честном планировании процесс получает долю пропускной способности диска, соответствующую его весу.

В алгоритмах с распределением бюджета BFQ [1] и MBFQ, описание которого приводится в данной работе, планировщик отдает каждому процессу диск на некоторое время, т.е. в течение некоторого периода времени на диске исполняются запросы только от этого процесса. Назовем такие периоды периодами активности, а процесс, запросы которого исполняются на диске в данный момент – активными.

Все планировщики дисковой активности стремятся, чтобы пропускная способность диска распределялась между процессами так же, как в *идеальной системе* [5]. Идеальной системой называется система, в которой дисковая активность исполняется непрерывно и четко в соответствии с весами процессов и в которой несколько процессов могут одновременно исполнять свои запросы к диску.

В идеальной машине предполагается следующее:

- Запросы приходят не в виде пакетов, а в виде непрерывного потока данных.
- Одновременно диск может читать или писать данные от нескольких процессов.
- Каждый процесс получает долю максимальной пропускной способности диска, равную его весу.

Такая концепция носит название Generalized Process Sharing (GPS) [5]. К сожалению, существующие жесткие диски не отвечают этим требованиям, потому что дисковая активность исполняется пакетами, а не непрерывно. Тем не менее, планировщики дисковой активности позволяют в реальных условиях добиться того, что запросы исполняются не

намного позже, чем в идеальной машине, и что каждый процесс получает долю пропускной способности диска, равную весу.

3. Постановка задачи

ОС Windows – одна из самых популярных и используемых операционных систем. Однако на сегодняшний день в ней нет дискового планировщика, который бы обеспечивал качество обслуживания приложений. Под обеспечением качества обслуживания подразумеваются гарантии, что каждый процесс получит свою долю пропускной способности диска в течение определенного промежутка времени (свойство честности [5]) и что каждый отдельно взятый запрос не будет ждать в очереди слишком долго.

До версии Windows 7 в ОС Windows применялся планировщик, переупорядочивающий запросы в соответствии с геометрией, а, начиная с Windows 7, появился так называемый приоритетный планировщик, который функционирует следующим образом: запросы ввода-вывода разделяются на три категории, согласно их приоритетности [7]. Самый высокий приоритет получают процессы, для которых критичны задержки в обслуживании, например, процесс медиа-плеера, и который нужно исполнить вне зависимости от других исполняемых процессов. Самый маленький приоритет имеют системные службы, которые предпочтительней исполнять во время простоя, чтобы они не задерживали обработку запросов от процессов, с которыми на данный момент работает пользователь. Планировщик следит, чтобы в первую очередь исполнялись запросы с более высоким приоритетом. Такой планировщик предотвращает ситуации, когда низкоприоритетные запросы занимают большую часть дискового ресурса, не давая исполняться запросам с большим приоритетом. Однако ввиду того, что большинство процессов имеет нормальный приоритет, значительная часть запросов никак не переупорядочивается, и качество обслуживания не повышается при использовании такого планировщика.

Теперь рассмотрим, каким образом отсутствие планировщика, обеспечивающего качество обслуживания, отражается на пользователях. Например, при одновременной работе программ, пишущих или читающих с диска, некоторые из них работают значительно медленнее, чем другие. Особенно заметно, если программа пользователя «тормозит» из-за того, что на фоне запустилось антивирусное ПО, проверяющее файлы. Другой пример: при запуске одновременно нескольких копирований файлов можно наблюдать, что хотя бы одно из них практически останавливается до тех пор, пока не закончатся другие. Также проигрывание видео- и аудио-файлов перестает быть плавным , если с диском параллельно работают еще несколько программ.

При использовании Windows на серверных системах и в контейнерной виртуализации отсутствие планировщика заметно еще сильнее. Если дисковый ресурс распределяется нечестно между процессами, обслуживающими клиентов, или контейнерами, то некоторым клиентам придется ждать ответа от сервера гораздо дольше, чем другим клиентам. Это критично в случае видео-серверов и контейнеров, где информацию от сервера требуется получать с минимальными задержками. Осложняет ситуацию и то, что клиенты платят за использование контейнеров и могут быть недовольны тем, что с контейнер медленно работает, потому что ему не досталось пропускной способности диска.

Итак, на данный момент в ОС Windows отсутствует дисковый планировщик, который бы помог решить описанные выше проблемы. Также отсутствуют какие-либо разработки по этой теме ввиду сложного устройства дисковой подсистемы ОС Windows и закрытости ее кода.

Задача данной работы – разработать и реализовать планировщик дисковой активности, который бы решал задачу обеспечения качества обслуживания приложений.

4. Существующие решения для планирования I/O

Существует множество решений для планирования ввода-вывода в разных областях. В данном разделе будут рассмотрены как самые простые решения, так и сложные, осуществляющие практически честное распределение ресурсов между пользователями, процессами или запросами.

4.1. Алгоритмы, учитывающие движение головки жесткого диска

При подготовке пункта 4.1 использовался материал [8] и [9].

- **SCAN**

Самый простой алгоритм. Его суть состоит в следующем: планировщик выстраивает запросы в очереди так, чтобы головка жесткого диска при их исполнении шла только в одном направлении. Другими словами, запросы переупорядочиваются в соответствии с их порядком на диске. При этом минимизируются передвижения головки, однако время отклика для конкретного запроса может быть очень большим, если все время будут приходить новые запросы с более «удобным» расположением на диске.

- **Shortest Seek Time First (SSTF)**

Напоминает предыдущий алгоритм. Здесь после выполнения каждого запроса планировщик выбирает новый запрос, который лежит на диске ближе всех к текущему.

- **Shortest Access Time First (SATF)**

Планировщик выбирает запрос с минимальной задержкой на поворот диска.

SATF и SSTF страдают от проблемы под названием «голодание» (starvation), то есть некоторые запросы могут очень долго ждать в очереди на обслуживание. Эту проблему решает SATFUF.

- **Shortest Access Time First With Urgent Forcing (SATFUF)**

Является модификацией SATF, но добавляется условие: если со времени прихода запроса прошел определенный промежуток времени, то запрос надо немедленно исполнить. При этом запросы, находящиеся на диске до него, можно переупорядочивать, если это не будет задерживать исполнения «старого» запроса.

SATFUF обладает рядом недостатков: во-первых, когда планировщик собирается найти и выполнить устаревший запрос, по пути удается переупорядочить и обработать мало запросов; во-вторых если один раз была запущена процедура исполнения устаревшего запроса, то издержки на это будут чаще всего очень большими, что приводит к превращению алгоритма в FCFS, т.к. тогда все запросы будут постепенно устаревать и обслуживаться вне очереди.

- **Shortest Access Time First with Urgent Forcing (N)**

Выбираются N старых запросов и переупорядочиваются (Здесь N – константа). Также можно переупорядочить и выполнить запросы, встречающиеся по пути, если это не задержит исполнение устаревших запросов.

- **Weighted Shortest Time First**

В этом алгоритме предполагаемое время, которое будет затрачено на операцию ввода, умножается на величину веса W. Вес вычисляется из того, сколько времени остается до deadline данного запроса. Время поиска с учетом веса определяется по формуле $T_W = T_{real}\ \frac{M-E}{M}$, где T_W -- время с учетом веса, T_{real} – предполагаемой время обслуживания IO запроса (запросы переупорядочиваются в порядке увеличения времени T_W), M – deadline запроса, E – время, прошедшее с момента прихода запроса в очередь.

Данный алгоритм дает среднее время обслуживания запроса, приблизительно равное аналогичному значению для SSTF, но максимальное время обслуживания в WSTF значительно меньше.

- **Aged Shortest Access Time First (ASATF)**

Также решает проблему starvation. В нем запросы обслуживаются в соответствии с величиной $M = wT_{age} - T_A/t$, где w – константа, обозначающая вес (в секторах в секунду), T_{age} – время, которое запрос провел в очереди, T_A -- время доступа к блоку на диске для этого запроса.

Этот алгоритм дает гарантии на то, что запрос будет обслужен, то при этом вновь пришедшие процессы могут ждать в очереди момента, пока их обслужат, довольно долгое время.

- **LOOK**

Алгоритм LOOK похож на алгоритм SCAN. В этом алгоритме запросы также переупорядочиваются в соответствии с расположением на диске, чтобы головка диска шла только в одном направлении. Но, в отличие от SCAN, этот алгоритм также отслеживает, есть ли еще запросы по направлению движения головки диска. Если запросов нет, то головка начинает движение в обратном направлении, обслуживая запросы, находящиеся в этом направлении. Таким образом, головка не доходит до конца диска, если там не находятся никакие запросы.

- **C-LOOK**

Одним из вариантов LOOK является C-LOOK. В алгоритме C-LOOK головка движется только в одном направлении. Когда по направлению движения головки больше нет запросов, головка возвращается в начало диска. Эффективность данного алгоритма связана с тем, что многие диски могут быстро переместить головку диска с последней дорожки на нулевую, причем даже за меньшее время, чем время поиска одной дорожки.

Данные алгоритмы являются самыми простыми из возможных ввиду отсутствия ограничений на время исполнения запроса. Если же такое ограничение присутствует, то необходимо применять более сложные, например, честные, алгоритмы.

4.2. Алгоритмы для систем реального времени.

Системами реального времени называются системы, в которых каждый запрос имеет свой крайний срок исполнения. Например, мультимедиа-системы являются системами реального времени. Приведем несколько алгоритмов, позволяющих соблюсти крайние сроки исполнения запросов.

Введем следующие понятия:

- r_i – минимальное время, когда запрос можно начать исполнять (например, момент его прихода в очередь)
- d_i – самое позднее время, к которому запрос может быть завершен.
- e_i – время, когда запрос начал исполняться.
- f_i – время, когда запрос фактически закончил исполняться.

Чтобы удовлетворить ограничения для real-time систем, должно быть выполнено $r_i \leq e_i$ и $f_i \leq d_i$.

Можно выделить два вида таких систем: строгие (в которых невыполнение dealine приводит к катастрофическим последствиям) и нестрогие (в которых несоблюдение deadline приводит к небольшим неудобствам работы системы.

- **Алгоритм ED**

Самое простое, что можно придумать для таких систем – обслуживание запросов в порядке их крайнего срока исполнения.

- **Алгоритм P-SCAN**

Алгоритм есть модификация SCAN, но с учетом приоритетов. Каждому запросу присваивается значение приоритета. Внутри каждого уровня приоритета алгоритм работает как обычный SCAN, т.е. обслуживаются все запросы данного уровня приоритета, встречающиеся на пути. Когда в данном направлении не остается запросов данного уровня, планировщик проверяет, нет ли запросов более высокого уровня. Если есть, то переходим на более высокий уровень. Если нет, то на более низкий.

- **SSEDO (Shortest Seek and Earliest Deadline by Ordering)**

Каждому запросу в очереди присваивается определенный вес. Затем вычисляется значение приоритета, равное весу, умноженному на расстояние между текущим блоком и блоком запроса, для которого вычисляется значение. Запросы обслуживают в порядке приоритета. Если у двух запросов одинаковый приоритет, то первым обслуживается запрос с меньшим крайним сроком исполнения.

В этом алгоритме те запросы, крайний срок исполнения для которых уже близко, получат высокий приоритет и будут обслужены раньше других.

Данный алгоритм плохо работает, например, в следующей ситуации: если крайний срок исполнения запроса A уже близок, а крайний срок исполнения запроса B далеко, но блок запроса B находится ближе, чем блок запроса A, то первым будет обслужен B. В это же время запрос A может уже выйти за свой крайний срок исполнения.

- **SSEDV (Shortest Seek and Earliest Deadline by Value)**

Алгоритм в некотором роде есть модификация SSEDO. Запросы обслуживаются в порядке возрастания их приоритета, который определяется как $a * d + (1 - a) * l$, где a – некий параметр, $0 <= a <= 1$, d – расстояние от текущего блока до блока запроса, l – оставшееся время до deadline.

SSEDO и SSEDV дают значительное улучшение производительности по сравнению с простейшими алгоритмами.

4.3. Алгоритмы для мультимедиа

При подготовке пункта 4.3 использовались материалы [10], [11], [12] и [13].

В большинстве алгоритмов периодические запросы исполняются в порядке истечения критического срока их исполнения. Как уже упоминалось, мультимедиа системы являются системами реального времени. В предыдущем разделе уже описаны некоторые решения для систем реального времени. Добавим в этот список описание алгоритмов, более подходящих для мультимедиа.

4.3.1. Алгоритм SCAN-EDF

SCAN-EDF – модификация описанного выше алгоритма SCAN. Запросы обслуживаются в порядке их deadline. Но если у запросов один и тот же крайний срок исполнения, то они обслуживаются в порядке их расположения на диске. Таким образом, этот алгоритм дает выигрыш только в системах, где часто несколько запросов имеют один и тот же крайний срок исполнения. Однако можно применить хитрый прием, увеличивающий количество запросов с одинаковым крайним сроком исполнени. Например, можно устанавливать крайние сроки исполнения, кратные определенным p.

Опишем алгоритм в виде псевдокода:

1) Пусть T -- набор из запросов с самыми близкими крайними сроками исполненя.
2) Если |T|=1 (т.е. в наборе один запрос), то обслужим его. Иначе пусть t_1 будет первым запросом в направлении сканирования. Обслужим t_1.
3) Переходим на шаг 1.

Направление сканирования выбирается в соответствии с CSCAN, т.е. выбирается тот запрос, который ближе расположен на диске.

Deadline модифицируются следующим образом:

Пусть D_i -- deadline запросов, а N_i – номер запроса в порядке его местонахождения на диске относительно текущего запроса. Тогда $D_i := D_i + f(N_i)$, где f – функция, которая задает время, прибавляемое к deadline, исходя из номера N_i. Эти прибавки должны быть достаточно малы, чтобы выполнялось $D_i + f(N_i) > D_j + f(N_j)$, если $D_i > D_j$, и запросы i и j должны быть обслужены в порядке их положения на диске, если $D_i = D_j$. Можно взять $f(N_i) = N_i/N_{max}$ или $f(N_i) = \frac{N_i}{N_{max}} - 1$.

Как правило, периодические запросы нуждаются в ответе со стороны системы. Для этого система ввода-вывода создает буфер, в который кладет ответ для текущего запроса. Распределение места в буфере для каждого из процессов – также отдельная задача. Также имеет значение размер буфера. Если размер маленький, то каждый процесс может использовать маленькую часть буфера короткие промежутки времени. Если же буфер

большой, каждый процесс может использовать более значительную часть буфера на более долгие промежутки времени. К тому же большой буфер позволяет обслуживать большее количество процессов.

Управление критическими временами также может положительно сказаться на производительности системы. К примеру, можно поставить крайний срок исполнения, являющийся суммой времени прихода запроса L и времени его обработки p. Однако если поставить крайний срок исполнения $L + 2p$, например, для двух запросов, то планировщик сможет успеть переупорядочить эти два запроса оптимальным образом. Таким образом, крайний срок исполнения лучше присваивать значение $L + m * p$, где m – некоторое число.

Стоит отметить, что для m запросов, ожидающих своей очереди, требуется место в буфере для хранения данных этих запросов, т.е. нужно как минимум $m * S$ свободного места в буфере. S – размер данных, хранимых одним запросом.

4.3.2. Deadline-Modification-Scan Scheme

Ниже приведен алгоритм, модифицирующий крайние сроки исполнения, чтобы найти оптимальный порядок исполнения запросов.

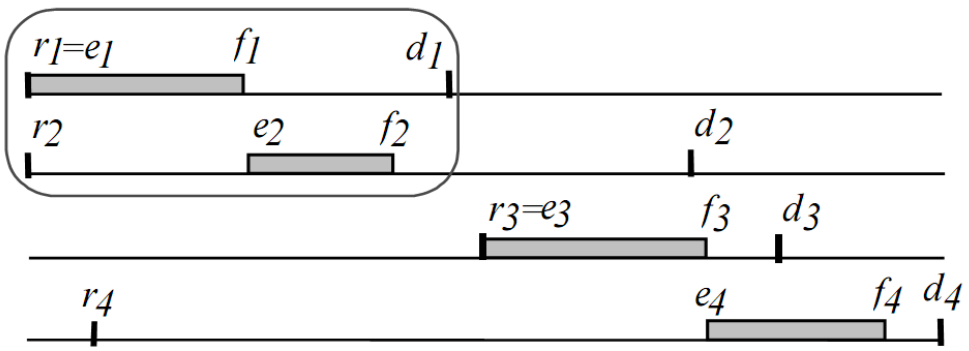
Пусть есть набор запросов $T = \{T_0, T_1, \dots, T_n\}$. Планировщик Z порождает очередность запросов $T = \{T_{Z(0)}, T_{Z(1)}, \dots, T_{Z(n)}\}$.

Будем называть Real-Time Disk scheduling (RTDS) планировщик Z, который набор $T = \{T_0, T_1, \dots, T_n\}$ преобразовывает $T = \{T_{Z(0)}, T_{Z(1)}, \dots, T_{Z(n)}\}$, где $f_{Z(n)}$ минимально, притом, что $r_{Z(i)} \leq e_{Z(i)}$ и $f_{Z(i)} \leq d_{Z(i)}$ для $i = 0 \dots n$.

Определим Максимальную сканируемую группу (MSG) следующим образом:

Пусть дан набор запросов, обработанный планировщиком $T = \{T_{Z(0)}, T_{Z(1)}, \dots, T_{Z(n)}\}$. Максимальная сканируемая группа G_i, начинающаяся с запроса T_i, есть максимальная группа $G_i = \{T_{Z(i)}, T_{Z(i+1)}, \dots, T_{Z(i+m)}\}$, удовлетворяющая $f_{E(k)} \leq d_{E(i)}$ и $r_{E(k)} \leq e_{E(i)}$ для $k = i \; to \; i + m$.

Алгоритм поиска MSG:



1) Дан набор $\{T_0, T_1, \dots, T_n\}$.
2) $K := 1;$
3) $For\ i \coloneqq 0\ to\ n\ do\ begin$

   $While\ \big((r_k \leq\ e_i)\ and\ (f_k \leq\ d_i) and\ (k \leq n)\big) do\ k \coloneqq k + 1;$

   $Start\_G_i$:=i;

   $End_{G_i} \coloneqq k - 1;$

   End

Поиск MU_MSG (mutually exclusive and un-scanned MSG groups)

1) $Z[0] :=\ 0;$ // $Z[i]$ есть направление сканирования G_i
2) $for\ k \coloneqq 1\ to\ n\ do\ begin$

   $If\ (a_{k-1} \leq\ a_k)\ then\ scan\_direction \coloneqq 1;\ else\ scan\_direction : =\ -1;$

   /* MSG просканирована тогда и только тогда, когда scan_direction есть +1, … -1 или -1, … -1 */

   $Z[k] :=\ Z[k-1] +\ scan\_direction\ ;$

   end
3) $i := 0;$
4) $while\ (i < n) do\ begin$

   $If\ ((\ |Z[End\_G_i] - Z[Start\_G_i]|\ +1) = size\ of\ G_i)$

   $/*\ если\ G_i\ оптимизирована\ по\ поиску\ */$

   $then\ i \coloneqq i + 1;\ /*\ следующая\ группа\ */$

   $else\ begin$

   $i :=\ End\_G_i\ + 1\ /*\ следующая\ группа\ */$

   end

   End

Deadline-Modification-Scan алгоритм для MSG

В этом алгоритме для каждой пары запросов T_iT_j модифицируются deadline $d_i = \min\{d_i, d_j\}$, чтобы удовлетворить требованиям EDF планировщика $d_i \leq d_j$. Это и есть deadline modification.

1) Store $d'_k :=\ d_k$ для каждого T_k
2) $repeat$

   $For\ k :=\ n - 1\ down\ to\ 1\ do\ /*\ deadline\ modification\ */$

   $if\ (d_k >\ d_{k+1})\ then\ d_k :=\ d_{k+1};$

   Найти все MSG группы для входящего набора запросов

Найти все MU_MSG группы из MSG групп.

$For\ i := 1\ to\ n\ do$

$If\ (G_i\ is\ MU\_MSG)\ then\ reschedule\ G_i\ by\ SCAN;$

$until\ (no\ deadline\ is\ modified)$

3) Recover $d_k := d'_k$ *для всех* T_k // восстанавливаем начальные deadline.

4.4. Честный алгоритм: Deficit Round-Robin

При подготовке пункта 4.4 использовался материал [14]

Примером честного алгоритма может служить Deficit Round-Robin. Он схож с Round-Robin, но в то же время ограничивает объем исполняемых запросов для каждого процесса.

Пусть f_i -- квота, данная процессу с номером i. Q_i – некоторая константа (квант). Пусть $Q = Mini\ (Qi)$. Определим f_i как $fi = Qi / Q$. Введем также переменную DC_i . Путь вначале DC_i равно Q_i для всех процессов. Алгоритм обходит все процессы по кругу. Если в

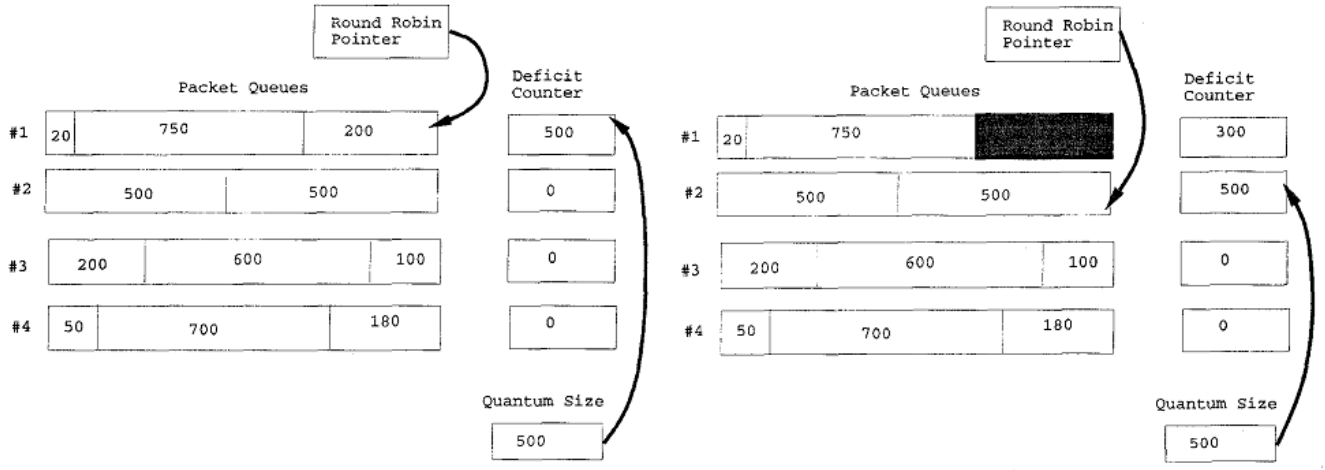


Рисунок 1

очереди процесса есть пакет, размер которого меньше DC_i , то этот пакет убираем из очереди, отправляем на обработку, а значение DC_i уменьшаем на величину размера пакета. Если размер пакета больше DC_i , пропускаем этот процес и прибавляем к DC_i значение Q_i.

Deficit round-robin – честный алгоритм, у которого выбор следующего процесса и следующего исполняемого запроса осуществляется за $O(1)$.

4.5. Планировщики в системе Linux

При подготовке пункта 4.5 были использованы источники [6] и [15].

Рассмотрим, какие планировщики имеются в ОС Linux.

В Linux используется 4 планировщика:

- NOOP
- Dealine.
- Anticipatory.

- Completely Fair Queeing (CFQ) – планировщик по умолчанию.

Рассмотрим их подробнее:

4.5.1. Noop (no operation – с отсутствием операций)

Наиболее простой из перечисленных.

Данный планировщик в соответствии с названием практически ничего не делает. Он обрабатывает поступающие запросы по принципу FIFO. Noop выполняет только объединение приходящих запросов со смежными.

Предназначен для работы с ОЗУ, флэш-носителями или устройствами, уже имеющими свой планировщик. К примеру, у плат флеш-памяти нет особых затрат на поиск, поэтому нет необходимости применять специальные алгоритмы вставки или объединения.

4.5.2. Лифтовый алгоритм Линуса (Linus elevator)

Это основной планировщик в версии 2.4. Предназначен для того, чтобы минимизировать поиск по диску в запросах очереди в целом, чтобы в конечном итоге головка диска проходила по диску практически по прямой.

Метод заключается в следующем: когда запрос добавляется в очередь, он сравнивается со всеми ожидающими запросами, чтобы найти подходящую пару для объединения. Затем определяется тип объединения: если найденный запрос предшествует только что пришедшему, что происходит добавление в конец запроса (back merging), а если следует за ним, то добавление в начало запроса (front merging). Если подходящей пары не найдено, планировщик вставляет новый запрос в очередь в позицию, которая наилучшим образом соответствует ему по номер сектора. Если и такой позиции не найдено, запрос помещается в конец очереди. Чтобы избежать ситуации, когда вновь прибывшие запросы не дают выполняться более старым из-за того, что больше подходят по номеру сектора и становятся в очередь раньше, было введено еще одно условие: если в очереди есть достаточно старый запрос, то вновь пришедшие запросы добавляются в конец очереди.

Однако и такой метод не позволяет полностью избежать задержек: может возникнуть ситуация, когда поступает много запросов к одному и тому же участку диска, при этом запросы к другим участкам практически не обрабатываются. Такой алгоритм не может обеспечить общедоступность и честность распределения ресурсов.

Более того, возникает проблема задержки обслуживания чтения при обслуживании записи. Когда ядру приходит запрос на запись, эта операция может начать выполняться не сразу, а тогда, когда это удобно ядру. Это связано с тем, что операция записи выполняется асинхронно по отношению к приложению, т.е. оно не ждет окончания процесса записи. Однако при операции чтения приложения имеют обыкновение зависать до окончания

последней, что существенно сказывается на производительности системы. По этой причине время задержки исполнения операции чтения важно минимизировать, в то время как время задержки операции записи не столь важно.

Положение усугубляет и тот факт, что часто приложение выполняет несколько последовательных операций чтения размером с буфер. При этом приложение ждет выполнение каждой из операций чтения. Это приводит к сложению задержек чтения и влечет за собой недопустимо большое время ожидания в целом.

4.5.3. Dealine – планировщик ввода-вывода с лимитом времени

В основном данный планировщик работает так же, как алгоритм Линуса (он тоже поддерживает очередь запросов в отсортированном по номерам секторов состоянии). При поступлении нового запроса Deadline аналогично Linus elevator осуществляет объединение и вставку запросов в очередь.

Однако цель этого планировщика – минимизировать задержки I/O операций. Для этого вводится еще две очереди запросов -- FIFO очередь запросов чтения и FIFO очередь запросов записи. Когда приходит новый запрос, он помещается в одну из двух очередей в зависимости от его типа. В отличие от основной очереди, очереди чтения и записи отсортированы по времени поступления запроса, а не по секторам диска. В них введено понятие времени ожидания (expiration time). По умолчанию это 500 миллисекунд для операций чтения и 5 секунд для операций записи.

Алгоритм Deadline планировщика заключается в следующем: запросы берутся из основной очереди и помещаются в очередь диспетчеризации, из которой они отправляются непосредственно к диску. Если для запроса чтения или записи в FIFO очередях истекает время ожидания, то начинают выполняться запросы из соответствующей очереди.

Таким образом, Deadline дает большее предпочтение операциям чтения, чем операциям записи, причем мы можем гарантировать, что через фиксированное время операция будет отправлена на обработку. Также мы можем гарантировать, что операции записи не будут препятствовать выполнению операций чтения, т.е. минимизируются важные для производительности задержки операций чтения.

Такой алгоритм подходит для систем, где количество операций чтения превышает количество операций записи, например, базы данных. В операционных системах такой алгоритм малоприменим. Дело в том, что из-за большого приоритета чтения в ситуации, когда разные запросы на чтение приходят один за другим, выполнение последних запускается сразу же. Для каждой операции чтения запускается операция поиска. Таким образом, большой приоритет операций чтения приводит к малым возможностям по оптимальному упорядочиванию запросов, что влечет большие задержки.

4.5.4. Прогнозирующий планировщик (Anticipatory I/O scheduler)

Предназначен для минимизации задержек чтения и во том же время обеспечением хорошей общей производительности. Данный планировщик, как и Deadline, использует три очереди запросов и учитывает время ожидания каждого из них. Отличительной же особенностью его является использование эвристического прогнозирования (anticipation heuristic), направленного на исключения обилия поступающих друг за другом операций чтения.

Алгоритм заключается в следующем:

Когда планировщик получает запрос на чтение, он его выполняет. Далее anticipatory ждет некоторое время (около 6 миллисекунд). Наиболее вероятно, что за это время придет еще несколько запросов на чтение. Вместо того, чтобы как Deadline, отправлять на выполнение практически следующий непосредственно за ним запрос на чтение, anticipatory выбирает из пришедших за 6 миллисекунд запросов те, которые обращены к соседним секторам диска, и отправляет их в очередь диспетчеризации. Таким образом, он осуществляет упорядочивание запросов так, чтобы минимизировать скачки головки по диску.

В результате если за время ожидания придут запросы к соседним секторам, затраты на ожидания окупаются за счет предотвращения двух лишних операций поиска. Если же за это время не было никакой дисковой активности к соседним секторам, то время ожидания просто теряется.

Более того, планировщик ведет статистику операций ввода-вывода по каждому процессу. Анализируя полученные данные, он пытается предсказать действия приложения. При удаче в предсказаниях, планировщик может существенно снизить затраты на поиск, а также больше уделить внимание тем запросам, которые хуже всего сказываются на производительности системы.

Недостаток планировщика в том, что задержки каждого конкретного процесса могут быть непростительно велики.

Данный планировщик хорошо работает на большинстве систем. Однако для некоторых систем, рассчитанных на большое количество поиска (например, серверы баз данных), anticipatory не дает преимущества.

4.5.5. Completely Fair Queeing (CFQ)

Является планировщиком по умолчанию начиная с версии 2.6.9. Как следует из названия, этот планировщик является честным.

Принцип работы CFQ в корне отличается от упомянутых выше. В данном планировщике для каждого процесса, отправляющего запросы ввода-вывода, создается

отдельная очередь, куда и складываются запросы от соответствующего процесса. Внутри каждой очереди запросы объединяются и сортируются. Таким образом, CFQ, как и другие планировщики, поддерживает свои очереди в упорядоченном состоянии.

Далее, CFQ берет из одной из очередей несколько запросов (по умолчанию 4), которые и отправляет на исполнение. Затем планировщик переходит к следующей очереди. Так он обходит очереди всех процессов по кругу.

Преимуществом такого планировщика в том, что можно гарантировать отсутствие больших задержек для каждого конкретного процесса. Это особенно важно для мультимедийных приложений.

Алгоритм подходит для систем, где многим приложениям требуется доступ к диску, как, например, в привычных нам операционных системах. Стоит отметить, что он позволяет получить хорошую производительность на разных типах нагрузки.

5. Алгоритм BFQ

В качестве первого этапа данной работы было решено реализовать для ОС Windows один из существующих планировщиков и исследовать его преимущества и недостатки. В ходе данной работы для реализации в Windows был выбран алгоритм Budget Fair Queuing (BFQ). Это наилучший честный алгоритм планирования на данный момент, несмотря на все его недостатки.

Алгоритм BFQ относится к классу честных. Он предоставляет гарантии того, что каждый процесс получит долю пропускной способности диска, независимо от поведения других процессов. Для BFQ доказано, что при его использовании запросы исполняются не намного позже, чем в идеальной машине, и что каждый процесс получает долю пропускной способности диска, равную весу.

BFQ разрабатывался как замена CFQ – планировщику Linux по умолчанию на настоящий момент. По результатам тестов BFQ показывает более хороший результат на чтении файлов, чем CFQ. Скорость чтения с BFQ была выше, чем с CFQ, и отклонение битрейта от среднего значения не превышало 3% против 28% у CFQ в условиях одновременного запуска нескольких процессов, исполняющих дисковую активность. В частности, с BFQ вещающий VLC видеосервер смог обслужить 24 параллельных потока без ощутимой потери пакетов, против 15 с CFQ [4]. В марте 2014 было объявлено, что разработчики готовят код для слияния с основной веткой ядра Linux [3].

BFQ работает на основе распределения бюджетов. Бюджет – количество байт, которые может прочитать или записать процесс за один раз. Когда у процесса появляются запросы к диску, BFQ присваивает ему некоторое значение бюджета. BFQ выбирает какой-либо

процесс из списка тех, у кого есть запросы к диску (в дальнейшем будем называть его активным процессом). Этот процесс будет исполнять активность в течение следующего промежутка времени. В соответствии с BFQ, в каждый момент времени работать с диском может только один процесс. Если бюджет процесса израсходован или у процесса больше нет запросов к диску, процесс прекращает быть активным, и BFQ выбирает следующий активный процесс.

Рассмотрим подробнее, как работает алгоритм BFQ. Первым шагом является выбор процесса, который будет следующим исполнять свои запросы, т.н. активного процесса. Для этой цели BFQ использует алгоритм Worst-case Fair Weighted Fair Queuing (WF^2Q+) [16] [17] [18], , который осуществляет выбор активного процесса на основе его веса и отметок времени.

Алгоритм WF^2Q+ опирается на понятие идеальной системы. Алгоритм выбирает следующий активный процесс из тех процессов, запросы которых уже начали бы исполняться к этому моменту в идеальной системе, причем выбирает тот процесс, который раньше закончил бы свою деятельность в идеальной системе. Для этого в WF^2Q+ введено понятие виртуального времени. Виртуальное время измеряется в количестве исполненной на данном диске дисковой активности. Также для каждого процесса считается $start\_time$ и $finish\_time$ – время, когда начнут и закончат исполнение все запросы, находящиеся в очереди запросов данного процесса. $Start\_time$ и $finish\_time$ пересчитываются каждый раз, когда процесс попадает в очередь ожидающих, т.е. когда у него появляются запросы к диску. $Start\_time$ равно времени прихода последнего исполненного запроса от этого процесса (это помогает избежать проблемы обманчивого бездействия (deceptive idleness), о которой пойдет речь ниже). Finish time рассчитывается по формуле $finish\_time = start\_time + service/weight$, где $weight$—вес процесса, а $service$ – размер дисковой активности, которую процесс исполнит. Когда процесс попадает в очередь ожидающих, $service$ считается равным бюджету, т.к. это максимальное количество байт, которые процесс исполнит, когда WF^2Q+ выберет его в следующий раз. После того, как

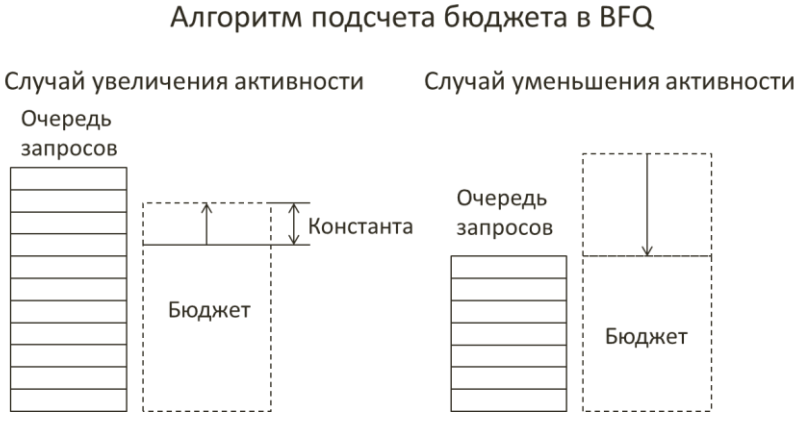


Рисунок 2

WF^2Q+ выбрал этот процесс, в случае если процесс израсходовал не весь свой бюджет, $finish\_time$ корректируется с учетом того, что величина $service$ составила меньше, чем значение бюджета.

После выбора активного процесса наступает стадия выбора бюджета для активного процесса. Эта и есть стадия, введенная BFQ, и отличающая его от WF^2Q+ в чистом виде. В BFQ алгоритм выбора бюджета достаточно прост: если запросы в очереди процесса кончились до того, как был полностью использован бюджет, то в следующем периоде активности бюджет процесса будет урезан на величину, равную неиспользованному бюджету. Если же после использования всего бюджета в очереди еще остались запросы, то в следующем периоде активности бюджет будет увеличен на некоторую константу B_{inc}, но не более чем максимальное значение бюджета.

После выбора величины бюджета запросы, помещающиеся в бюджет (т.е. сумма их размеров не превышает бюджет), отправляются к диску, и процесс перестает считаться активным. WF^2Q+ корректирует метки времени, если процесс использовал не весь свой бюджет, и выбирает следующий активный процесс. Запросам предыдущего процесса, которые не поместились в бюджет, придется ждать, пока WF^2Q+ выберет этот процесс следующий раз.

Сложность алгоритма WF^2Q+, лежащего в основе BFQ, – O(log N) в худшем случае, где N – количество процессов, конкурирующих за дисковый ресурс. Сложность операций, осуществляемых BFQ при приходе нового запроса или выборе активного процесса и работе с ним, – O(1). Таким образом, сложность BFQ составляет O(log N) в худшем случае.

6. Преимущества BFQ

Алгоритм BFQ построен таким образом, чтобы избежать многих проблем, встречающихся у других планировщиков.

6.1. Соблюдение гарантий честности

В статье [1] утверждается, что для любого интервала времени, в течение которого у процесса есть запросы к диску, выполнено следующее:

$$W_i \cdot S(t_1, t_2) - S_i(t_1, t_2) \leq B_{max} + B_{i,max} + L_{max} \qquad (1)$$

где W_i — вес i-го процесса,

$S(t_1, t_2)$ – количество байт, прочитанных или записанных на диск за интервал времени (t_1, t_2) в идеальной машине,

$S_i(t_1, t_2)$ – количество байт, прочитанных или записанных на диск за интервал времени (t_1, t_2) i-м процессом на реальной машине,

B_{max} – максимальный размер бюджета,

$B_{i,max}$ – максимальный размер бюджета для i-го процесса,

L_{max}—максимальный размер запроса.

Таким образом, первое преимущество BFQ состоит в том, что он гарантирует, записанное или прочитанное количество байт конкретного процесса в реальной машине $S_i(t_1, t_2)$ отличается от аналогичной величины $W_i \cdot S(t_1, t_2)$ в идеальной машине не более, чем на фиксированную величину, не зависящую от поведения других процессов. Это сильное утверждение. Учитывая, что другие планировщики не могут дать такие гарантии, можно назвать BFQ наиболее честным из существующих планировщиков.

Примечательно также, что BFQ гарантирует, что процесс получит свою долю пропускной способности диска вне зависимости от его бюджета. С одной стороны, процесс с большим бюджетом долго исполняет свою дисковую активность, не давая другим процессам исполнять свои запросы. В этот промежуток времени честность нарушается. Однако процессы с большим бюджетом будут редко выбираться BFQ в качестве активных, поэтому в среднем на больших интервалах времени честность сохраняется.

Один из лучших честных планировщиков на сегодняшний день – CFQ, дисковый планировщик Linux по умолчанию. Как и BFQ, он отдает диск в пользование только одному процессу (активному), что помогает увеличить суммарную пропускную способность диска, т.к. головке диска приходится меньше перемещаться между запросами на диске . Однако CFQ отдает диск процессу на фиксированный интервал времени, а не на фиксированное значение бюджета, как BFQ. Это может сильно сказываться на честности, т.к. за один и тот же интервал времени процессы могут записать или прочитать абсолютно разное количество байт. В частности, если процесс читает из разных мест, удаленных друг от друга, то большая часть времени может уйти только на перемещение головки диска. CFQ обслуживает процессы по алгоритму Round-Robin. Это означает, что в худшем случае запросы будет исполнены с задержкой O(N), где N – количество процессов, желающих получить доступ к диску. Задержка, зависящая от количества процессов, недопустима в пользовательских системах, т.к. с ростом числа процессов задержки неограниченно растут.

6.2. Гарантии задержек исполнения запросов

Вторым существенным преимуществом BFQ является его гарантии по времени. Утверждается, что для любого пришедшего запроса промежуток времени от момента его прихода до начала его исполнения не превышает $d_{i,max}$:

$$d_{i,max} \stackrel{\text{def}}{=} max_j d_i^j \leq \frac{B_{i,max} - min_j L_i^j}{W_i b_{agg}} + \frac{B_{max} + L_{max}}{b_{agg}} \quad (2)$$

где L_i^j – размер j-го запроса в очереди i-го процесса,

d_i^j – задержка j-го запроса в очереди i-го процесса,

b_{agg} – суммарная пропускная способность диска.

Как нетрудно видеть, значение $d_{i,max}$ не зависит от времени и от поведения других процессов и не накапливается со временем. Так же, как и гарантии честности, это очень хороший результат для дискового планировщика. В Linux ввиду ограничений на максимальный размер запроса в системе L_{max} и относительно маленьких значениях B_{max} (128 Кб) значение максимальной задержки $d_{i,max}$ не превышает 0,7 секунды, что удовлетворяет требованиям аудио и видео приложений.

В то же время $d_{i,max}$ зависит от максимального бюджета, присвоенного i-му приложению. Аудио и видео приложения выполняют маленькие периодические запросы, что позволяет присвоить им маленькое значение бюджета. Поэтому максимальная задержка для таких приложений маленькая, что позволяет аудио и видео приложениям работать без задержек независимо от активности других приложений. Это свойство важно для пользователей.

Аналогичным свойством обладают real-time планировщики, которые переупорядочивают запросы на основе отметок времени. Однако они зачастую не могут дать как гарантии честности, так и высокую суммарную пропускную способность диска.

6.3. Проблема отложенного прихода (delayed arrival)

Проблема отложенного прихода или delayed arrival касается планировщиков, которые распределяют запросы на основе отметок времени (timestamp). В момент, когда запрос приходит к планировщику, тот может быть занят, например, обработкой другого запроса. В этом случае планировщик обработает событие о том, что пришел новый запрос, позже, чем запрос фактически пришел, и, как следствие, поставит ему более позднюю отметку времени. Таким образом, запрос начнет и закончит исполняться позже, чем должен был в соответствии с честным распределением диска. Если запрос синхронный, то следующий за ним синхронный запрос также придет позже. В результате битрейт этого процесса снижается, и гарантии битрейта нарушаются.

В BFQ эта проблема решается следующим образом: если запрос приходит от активного приложения, то его отметка времени его прихода считается равной времени окончания предыдущего исполненного запроса. Таким образом, отметки времени сдвинуты назад по сравнению с текущим моментом времени и более корректно отражают время прихода данного запроса [1].

6.4. Проблема обманчивого бездействия (deceptive idleness)

В алгоритме BFQ учтено, что если процесс посылает синхронные запросы, то планировщику не придет новый запрос, пока не исполнится предыдущий. При этом сразу после того, как запрос отправлен диску, планировщик может ошибочно посчитать, что процесс больше не хочет исполнять дисковую активность, и выбрать в качестве активного другой процесс. Для того чтобы избежать такой ситуации, в BFQ введен интервал времени T_{wait} . Когда последний запрос из очереди irp процесса был отправлен диску, BFQ ждет в течение T_{wait} , не придет ли планировщику новый запрос от этого процесса. Это очень эффективный способ увеличить битрейт процесса с синхронными запросами, не смотря на то, что диск простаивает во время ожидания T_{wait} . Во-первых, BFQ не приходится выбирать новый активный процесс после исполнения каждого синхронного запроса. Во-вторых, каждый выбор активного процесса часто сопровождается перемещением головки диска на большое расстояние. Исполняя больше запросов от одного процесса подряд, мы в большинстве случаев сокращаем расстояние, которое надо преодолеть головке диска. В-третьих, процесс не ждет, пока его снова выберут активным для исполнения следующего синхронного запроса. Т.е. повышается время ответа приложения. Однако к выбору времени T_{wait} нужно подходить осторожно, т.к. в этот интервал времени диск простаивает, и при большом T_{wait} можно существенно снизить пропускную способность диска.

6.5. Условия гарантий честности

Одной из особенностей BFQ является то, что бюджет, основной параметр, влияющий на время ответа приложения, не влияет на честность распределения дискового ресурса. Как видно из формулы (1), на гарантии четности влияют только максимумы бюджетов и размеров запросов. Это свойство будет использоваться в данной работе. Благодаря этому свойству, процессу можно присваивать любое значение бюджета, не превышающее максимум, не заботясь о соблюдении гарантий честности.

7. Недостатки алгоритма BFQ

Как и любой планировщик, BFQ обладает некоторыми недостатками и ограничениями применения. BFQ разрабатывался в первую очередь для машин с постоянной и высокой дисковой активностью, где честность распределения ресурсов важнее, чем время ответа приложения. BFQ хорошо справляется с поставленными задачами, однако на пользовательских машинах, где важно быстрое реагирование системы на действия пользователя, у этого планировщика заметны существенные недостатки.

7.1. Продолжительное ожидание синхронных запросов

В Windows введение параметра T_{wait} , спасающего BFQ от проблемы обманчивого бездействия или deceptive idleness, не так эффективно, как в Linux. Когда запрос исполнился, обработка события, что запрос исполнен, выполняется не сразу, а позже. Максимальная задержка исполнения этого обработчика составляет 16 мс. Таким образом, чтобы дождаться следующего синхронного запроса, нужно не только подождать, пока исполнится текущий запрос и процесс отреагирует на это событие, но и дождаться DPC. Стоит также учитывать, что у диска есть внутренний планировщик запросов и очередь запросов, поэтому нельзя гарантировать, что запрос начнет исполняться, как только BFQ пошлет его вниз. Для того, чтобы гарантировать, что через интервал времени T_{wait} процесс пришлет следующий синхронный запрос, нужно, чтобы T_{wait} удовлетворяло $T_{wait} \geq T_{request\ execution} + T_{DPC\ completion} + T_{application\ answer}$, что как минимум больше 16 мс. Однако такое большое время простоя диска после каждого синхронного запроса негативно отражается на суммарной пропускной способности диска и времени ответа приложения.

7.2. Обратный эффект честности

Честность – основное свойство, которого мы стремимся добиться, разрабатывая планировщик. Однако не менее важным свойством для пользователя является время ответа приложений, ради которого можно временно пожертвовать честностью, и BFQ не учитывает данный факт. Один из наиболее часто встречающихся сценариев, где это заметно, -- запуск приложения. Приложению на старте нужно считать какие-либо данные, необходимые ей для дальнейшей работы. При этом другие процессы также конкурируют за дисковый ресурс. Эксперименты показывают, что на системе без планировщика приложение запустится гораздо быстрее, чем на системе с честным планировщиком, т.к. в последнем случае дисковая активность будет распределяться честно, и приложение получит только гарантированную ему долю пропускной способности. Такое поведение недопустимо на пользовательских системах.

7.3. Учитывает геометрию жесткого диска.

В алгоритме BFQ запросы внутри очереди каждого процесса переупорядочиваются в соответствии с алгоритмом C-LOOK. Цель такого приема – расположить запросы в очереди так, чтобы головка диска совершала как можно меньше перемещений, и запросы исполнялись быстрее. Однако такой подход совершенно не оправдывает себя. Все современные жесткие диски имеют внутреннюю очередь, в которой запросы переупорядочиваются с учетом геометрии (технология NCQ [19] [20] для SATA устройств). Таким образом, разрабатывая планировщик дисковой активности, мы избавлены от необходимости заботиться о геометрии. Стоит упомянуть, что переупорядочивание запросов во внутренней очереди диска может влиять как на гарантии честности, так и на

задержки выполнения запросов, однако это проблема касается всех планировщиков, а не только BFQ.

7.4. Стремление сократить бюджет процессу

Если процесс не расходует весь бюджет за период активности, то в следующий раз BFQ присвоит ему бюджет, равный количеству использованного бюджета в предыдущем периоде. С одной стороны это обусловлено тем, что чем меньше бюджет процесса, тем меньшие задержки будут испытывать как запросы данного процесса, так и запросы других процессов. С другой стороны, урезать бюджет, оставляя ровно столько, сколько было использовано в предыдущий раз, нерационально. Весьма вероятно, что в следующий раз процессу понадобится исполнить больше запросов, и учитывая медленную реакцию BFQ на увеличение дисковой активности, такой подход вызовет еще более заметное увеличение времени ответа приложения.

Рассмотрим один из сценариев, где проявляется этот недостаток. Допустим, процесс стабильно много читает или пишет. Однако в один из периодов активности он записал меньше, чем в предыдущие. К примеру, при копировании файлов приложению понадобилось спросить у пользователя, нужно ли заменять файл, и ждет реакции пользователя. BFQ определяет, что у приложения кончились запросы и урезает ему бюджет. Однако в следующем периоде активности приложению снова нужен большой бюджет. Как показано в следующем пункте увеличение битрейта этого приложения может происходить очень долго.

7.5. Медленная реакция на увеличение дисковой активности

Если процесс израсходовал весь свой бюджет, но в его очереди все еще остались запросы, то BFQ увеличивает ему бюджет на некоторую фиксированную величину (назовем ее B_{inc}) на следующий период активности. Подобный подход порождает главный недостаток BFQ, на который жалуются пользователи, – медленная реакция системы на увеличение дисковой активности приложения, в частности, медленный старт приложений. Этому способствуют следующие факторы:

- BFQ увеличивает бюджет приложения на B_{inc}, однако воспользоваться увеличенным бюджетом процесс может только в следующем периоде активности. Таким образом, запросам, не успевшим исполниться в предыдущий период активности нужно будет ждать порядка $\frac{(N-1)\bar{B}}{b_{agg}}$, чтобы начать исполняться. Здесь N – число процессов, исполняющих дисковую активность, $\bar{B}$ — средний бюджет приложений, b_{agg} – суммарная пропускная способность диска. Оценим это значение. Возьмем количество процессов порядка 10. Средний бюджет процесса порядка 4 * 10^6 байт, т.е. порядка максимального размер

запроса в ОС Windows (более подробно будет пояснено в пункте 9.7). Суммарная пропускная способность диска составляет $40 * 10^6$ байт/секунду.

$$\frac{(N-1)\bar{B}}{b_{agg}} \sim \frac{10 * 4 * 10^6}{40 * 10^6} = 1\ c$$

Ясно, что для пользовательского приложения задержки в 1 секунду существенны.

- Допустим, приложение резко увеличило свою дисковую активность. Это часто встречающая ситуация, например, когда приложение стартует или сбрасывает результаты своей деятельности на диск. Если до этого приложение исполняло мало дисковой активности, что его бюджет будет маленьким (см. пункт 6.4.). Тогда BFQ будет последовательно увеличивать бюджет этому процессу на B_{inc}, причем между каждыми увеличениями будет проходить время порядка $\frac{(N-1)\bar{B}}{b_{agg}}$. Так, если приложение хочет исполнить в этом периоде активности S байт, то последний запрос из S исполнится через n периодов активности, где n определяется следующим выражением:

$$S = B_i + (B_i + B_{inc}) + (B_i + 2B_{inc}) + \cdots = nB_i + \frac{n(n+1)}{2} B_{inc}$$

откуда n равно:

$$n = \frac{-B_i - \frac{B_{inc}}{2} \pm \sqrt{B_i^2 - B_i B_{inc} - \frac{3}{4} B_{inc}^2}}{B_{inc}}$$

Время ожидания исполнения последнего запроса составит порядка

$$n \frac{(N-1)\bar{B}}{b_{agg}}$$

Допустим, приложение записало или прочитало 100 Кб ($B_i = 10^5$) за предыдущий период активности. Потом ему понадобилось записать 10 Мб ($S = 10^6$). Пусть $B_{inc} = 0.5 * 10^6$. Тогда $n \sim 5$ и время ожидания составить~12.5 секунд.

- Если в очереди процесса после исчерпания бюджета осталось мало запросов (сумма их размеров может быть меньше B_{inc}), то бюджет все равно будет увеличен на B_{inc}, несмотря на то, что в этом нет необходимости.

Итак, из-за своего подхода к обработке увеличения дисковой активности BFQ страдает из-за больших времен задержек между приходом запроса и началом его исполнения, что приводит к высокому времени ответ пользовательских приложений.

7.6. BFQ не учитывает количество запросов в очереди процесса

BFQ не принимает во внимание, сколько запросов в очереди у процесса при подсчета бюджета. Планировщик учитывает только, был ли израсходован в прошлый раз весь бюджет, и если нет, то какая его часть была израсходована. Однако такой подход не верен.

Если в очереди процесса мало запросов, то нет причин давать ему большой бюджет. Если же их много, то имеет смысл дать большой бюджет, т.к. в противном случае процесс не выполнит не все запросы из очереди, и в следующий раз все равно его бюджет необходимо будет увеличить, но процессу нужно будет дождаться, пока его выберут активным в следующий раз (снова увеличивается время ответа приложения).

7.7. Невыполнение гарантий задержек

Гарантии задержек BFQ могут не соблюдаться при некоторых условиях.

- Допустим, у i-го процесса появились запросы в очереди и так случилось, что его параметр $finish\_time$ оказался меньше, чем у процесса, который активен в данный момент. В этом случае i-му процессу придется ждать, пока закончит исполнять свои запросы активный процесс, хотя в идеальной системе его запросы закончили бы исполняться раньше.
- Активный процесс не израсходовал весь свой бюджет. Это значит, что ранее этому процессу было присвоено значение $finish\_time$ больше, чем на самом деле, и, возможно, из-за этого он стал активным позже, чем должен был.

Условия, при которых гарантии задержек не выполняются, следуют из алгоритма выбора активного процесс WF^2Q+. В данной работе WF^2Q+ используется в неизменном виде, поэтому эти недостатки остаются.

8. Особенности реализации BFQ в ОС Windows

При реализации BFQ на ОС Windows возникли сложности, связанные в первую очередь с тем, что некоторые предположения BFQ не выполняются в реальных операционных системах. Также возникли проблемы ввиду того, что алгоритм предназначен в первую очередь для Linux, а I/O подсистема Windows в корне отлична от I/O подсистемы Linux.

Рисунок 3

8.1. Список процессов не постоянен

В реальных системах практически никогда не встречаются ситуации, когда в течение долгого времени с диском работают одни и те же процессы. Как правило, процессы обращаются к диску, затем прекращают свою активность, обращаются снова и т.д. Во время бездействия процесса (в плане дисковой активности) нет необходимости его держать в очереди активных процессов, пересчитывать для него *finish_time*, *start_time*, бюджет и другие параметры. В тоже время встает вопрос о том, как правильно обработать ситуацию, когда появляется новый инициатор, т.к. ему нужно

установить бюджет, достаточный для того, чтобы время ответа приложения оставалось приемлемым, но при этом не нарушалась значительно честность распределения дисковой активности для других приложений.

8.2. У процесса не всегда есть запросы к диску

BFQ гарантирует, что дисковая активность распределяется честно, предполагая, что у инициатора всегда есть запросы к диску, однако это не так. Возникает проблема, как обрабатывать часто встречающую ситуацию, когда у процесса заканчиваются запросы к диску и в течение долгого времени запросы от этого процесса больше не приходят.

8.3. В системе может быть несколько дисков

BFQ не подразумевает, что в системе может быть несколько дисков. При этом с каждым из них нужно обеспечить честность процессов и максимизировать суммарную дисковую активность. Самый простой способ достижения этой цели – дать каждому диску его собственный планировщик BFQ. Однако в связи с недостатками существующей системы мониторинга, на основе которой разрабатывался планировщик, тяжело сделать, чтобы каждый диск работал со своим собственным BFQ. Это повлекло за собой необходимость некоторой модификации BFQ, в которой для каждого инициатора есть данные, общие для всех дисков, с которыми он работает, и есть данные, уникальные для каждого диска. Данные, общие для всех дисков, – это вес процесса и его максимальный бюджет. Для каждого раздела у инициатора есть своя очередь запросов ввода-вывода, бюджет, *start_time* и *finish_time*.

8.4. Диск – черный ящик

В оригинальном BFQ функция dispatch() вызывается, когда предыдущий запрос к диску исполнился и диск запрашивает у планировщика следующий запрос, который нужно исполнить. Предполагается, что диск в каждый момент времени исполняет только один запрос. Когда процесс выбран BFQ как активный, только он исполняет запросы на диске. Однако такой подход не применим к современным HDD, т.к. они имеют собственную очередь запросов и собственный планировщик, переупорядочивающий запросы с целью минимизации перемещения головки диска (технологии NCQ [19] [20] и TCQ [21]). Больше нельзя гарантировать, что запрос начнет исполняться сразу же после того, как BFQ послал его вниз, и что в данный конкретный момент исполняются запросы только от одного процесса. К тому же, необходимо решить, когда вызывать функцию dispatch().

В модификации BFQ для Windows функция BFQProcessQueue (аналог функции dispatch()), выбирающая следующий активный процесс и посылающая запросы диску, вызывается, когда приходит новый запрос, когда заканчивается какой-либо запрос, либо по

таймеру через промежутки времени $T_{process\_queue}$. При каждом вызове BFQProcessQueue из очередей вынимается и отправляется диску N запросов. Число N не должно быть равно 1, как в оригинальном BFQ, т.к. это повредит суммарной пропускной способности диска. N также не должно быть слишком большим, т.к. иначе во внутренней очереди диска скопится много запросов и внутренний планировщик диска их переупорядочит в соответствии с геометрией, что отразится на гарантиях честности и времени ответа.

8.5. Внутренний планировщик диска.

Современные жесткие диски обладают собственными планировщиками запросов (технологии NCQ и TCQ). которые оптимизируют запросы с учетом геометрии диска. С одной стороны они был разработаны с целью улучшить производительность диска и должны дополнять функциональность планировщика в ОС, т.к тот переупорядочивает запросы исходя из соображений честности, а не геометрии. Однако иногда внутренний планировщик мешает работе планировщика в ОС. Рассмотрим, каких случаях это может происходить.

- BFQ посылает запросы на исполнение исходя из их размера и приоритета. NCQ же может отложить исполнения высокоприоритетного запроса и выполнять те, которые более удобно выполнить исходя из их местоположения на диске, тем самым нарушая гарантии задержки.
- После исполнения синхронного запроса BFQ ждет некоторое время, не придет ли следующий синхронный запрос, заставляя диск простаивать. В это время NCQ может начать исполнять другие запросы, чтобы избежать простоя диска. Это сводит на нет преимущества этого приема BFQ, так как во время ожидания следующего синхронного запроса головка диска уже могла переместиться в другое место на диске.

Влияние внутреннего планировщика на работу планировщика в ОС в данной работе не исследуется.

8.6. Некоторые запросы системы нельзя задерживать

Далее рассмотрим особенности, связанные с устройством подсистемы ввода-вывода в Windows.

В ОС Windows есть запросы, которые нельзя задерживать, иначе система выдаст синий экран. К таким относятся, например, обработка ядерного page fault, записи в журнал NTFS и сброс оперативной памяти на диск в условиях нехватки памяти (своп). Данные типы запросов в модификации BFQ для Windows идут в обход BFQ, попадая в очередь к диску напрямую. Если таких запросов слишком много, их исполнение может отразиться на гарантиях честности. Но, как правило, подобная системная активность (кроме сброса оперативной памяти) имеет маленькие размеры и не влияет на дисковую активность других

процессов. В случае же сброса оперативной памяти в условиях нехватки памяти ни о честности, ни о заботе о времени ответа приложения, речи уже не идет.

8.7. Запросы могут быть большими

В Linux по умолчанию максимальный размер запроса ограничен 512 Кб. В Windows нет максимума на размер запроса. При проведении тестов были зафиксированы запросы размером 4 Мб. В BFQ максимальная задержка для запроса зависит от максимального размера запроса (формула (1)), и есть в Windows размер запроса настолько большой, то гарантируемая максимальная задержка в BFQ оказывается слишком большой и не отвечающей требованиям на время ответа приложения.

Более того, от максимального размера запроса неявно зависит B_{max}. Если размер запроса окажется больше B_{max}, то он никогда не исполнится, а следовательно, и все остальные запросы от этого процесса, т.к. у процесса всегда будет не хватать бюджета для его исполнения. С другой стороны, можно сделать B_{max} достаточно большим, чтобы размеры запросов были всегда меньше этой величины. Тогда каждый процесс сможет за одну итерацию писать или читать B_{max} байт, а другие процессы будут в это время ждать.

Подсчитаем, какими будут гарантии BFQ в условиях больших B_{max} и L_{max}. Из тестов, проведенных на ОС Windows, L_{max} может достигать 4 Мб. Возьмем B_{max} = L_{max} и $B_{i,max} = B_{max}$. BFQ утверждает, что разница между количеством байт, прочитанных или записанных на диск i-м процессом в идеальной машине, и аналогичной величиной на реальной машине определяется следующим выражением:

$$W_i \cdot S(t_1, t_2) - S_i(t_1, t_2) \leq B_{max} + B_{i,max} + L_{max} \approx 6 \text{ Мб}$$

В этом контексте 6 Мб – очень большое число, и данное свойство теряет свой смысл.

Рассмотрим теперь выражение для максимальной задержки между приходом запроса и началом его исполнения. Возьмем в качестве максимальной пропускной способности 40 Мб/с.

$$d_{i,max} \equiv max_j d_i^j \leq \frac{B_{i,max} - min_j L_i^j}{W_i b_{agg}} + \frac{B_{max} + L_{max}}{b_{agg}} < \frac{B_{i,max}}{W_i b_{agg}} + \frac{B_{max} + L_{max}}{b_{agg}}$$

$$d_{i,max} \approx \frac{1}{W_i 10} + \frac{1}{5}$$

Если вес процесса W_i большой (например, 1/2), то задержка имеет весьма приемлемое значение. Однако если брать более реалистичный вес процесса (1/10 или 1/100 для больших систем), то задержка *каждого* запроса будет составлять от 1.5 секунд до нескольких секунд, что недопустимо для пользователя.

8.8. Запросы могут разбиваться на более мелкие

Еще одна особенность связана с тем, что файловая система может разбивать запросы на более мелкие и планировщика они доходят уже в раздробленном виде. В если реализовывать оригинальный BFQ в ОС Windows, то могут возникать ситуации, когда несколько частей синхронного запроса отправлены планировщиком на исполнение, а оставшимся BFQ не дал исполниться за текущую итерацию, потому что у процесса кончился бюджет. Следовательно, следующий синхронный запрос от этого процесса планировщик получит только после того, как этот процесс будет выбран снова, оставшиеся части синхронного запроса будут обработаны, и процесс обработает событие, что его дисковый запрос исполнен. Такой сценарий также снижает суммарную пропускную способность диска.

9. Модифицированный BFQ (MBFQ)

При реализации планировщика BFQ в ОС Windows было выявлено несколько существенных недостатков, мешающих работе пользователя. Поэтому в рамках данной работы была создана новая модель распределения бюджетов между процессами, учитывающая недостатки оригинального BFQ.

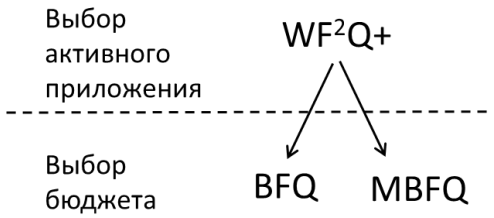


Рисунок 4

Модифицированный BFQ или MBFQ – планировщик, также использующий понятие бюджета ля управления дисковым ресурсом. Как и BFQ, данная модель использует WF^2Q+ для соблюдения гарантий честности. Однако здесь предложена совершенно иная модель распределения бюджетов между процессами, позволяющая улучшить время ответа приложения и сделать планировщик более подходящим для использования на машинах простых пользователей. Модифицированный BFQ предугадывает дальнейшее поведение процесса исходя из его истории запросов. Благодаря этому удается предсказать, сколько бюджета понадобится процессу в следующий период активности, чтобы дать процессу именно столько бюджета, сколько ему понадобится. В то же время MBFQ контролирует, чтобы бюджет процесса не был слишком большим, и процесс не задерживал исполнение запросов других процессов. Гарантии,

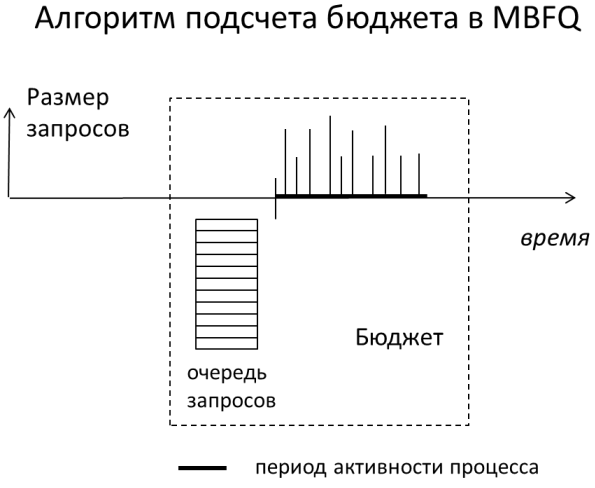


Рисунок 5

предоставляемые оригинальным BFQ, не зависят от значения бюджета процесса, только от его максимального значения, поэтому для MBFQ справедливы те же гарантии честности, что и для BFQ.

Сохраняя все преимущества BFQ, MBFQ исправляет самый заметный и раздражающий недостаток оригинального планировщика – медленную реакцию на изменение активности приложения.

9.1. Цели создания MBFQ

Сформулируем цели, которых хотелось бы добиться в алгоритме MBFQ.

1) Сохранение свойства честности. К счастью, свойство честности обеспечивается алгоритмом выбора активного процесса WF^2Q+, на котором базируется MBFQ, поэтому в MBFQ не нужно заботиться об этом.

2) Гарантированный битрейт, равный весу. Если процессу нужен битрейт, меньший либо равный его весу, планировщик обязан представить его вне зависимости от активности других процессов.

3) Битрейт процесса должен быть равен идеальному. Определим ошибку e_i битрейта для i-го процесса следующим образом:

$$e_i = b_i - b_i^{*W},$$

где b_i — битрейт i-го приложения, $b_i^{*,W}$ — идеальный битрейт i-го приложения с учетом веса. Задача MBFQ – минимизировать данную ошибку.

4) Улучшение времени ответа приложения: алгоритм должен давать процессу столько бюджета, сколько тот может израсходовать. Оценка нужного процессу бюджета – важная составляющая MBFQ. Размер бюджета сильно влияет на время ответа приложения. Если бюджет слишком мал, то активные процессы меняются чаще, и головке диска нужно чаще перемещаться, чтобы считать данные для другого процесса. Также если запрос большой, он может просто не поместиться в бюджет. Если же бюджет большой, то существенно увеличивается интервал времени между приходом запроса и фактическим началом его исполнения, т.к. приложению нужно дождаться, пока текущий активный процесс закончит свою деятельность.

5) Быстрая реакция на изменение активности приложения. Как указано в пункте 7.4 и 7.5, BFQ очень медленно реагирует на изменение активности, а именно на ее увеличение, что сказывается на времени ответа приложения. В MBFQ необходимо обеспечить быструю реакцию на увеличение количества запросов от приложения. Это свойство пересекается предыдущим: алгоритм даст процессу больше бюджета, если тот начал присылать больше запросов.

6) Суммарная пропускная способность диска должна быть как можно ближе к максимальной. Отметим, что максимальная пропускная способность диска достигается только если в течение долгого времени исполнять запросы от одного и того же процесса, особенно если процесс исполняет последовательные чтение или запись. Если же процесс исполняет случайные чтение или запись, то никакой планировщик не поможет достичь максимальной пропускной способности.

9.2. Предположения

В предложенном нами алгоритме используются следующие предположения:

1) Активность процесса в текущем периоде будет происходить так же, как и в прошлом (в плане времени прихода и размеров запросов)

2) За прошлый период T планировщик выбирал данный процесс примерно столько же раз, сколько в прошлом периоде. Данное предположение следует из предположения 1), т.к. в соответствии с WF^2Q+ планировщик выбирает приложение, основываясь на времени прихода и размере его запросов.

9.3. Схема работы планировщика

Поясним общую схему работы планировщика. Один цикл его работы включается в себя следующие этапы:

I) Стадия выбора активного процесса
   1) С помощью WF^2Q+ выбирается следующий активный процесс
   2) *Вычисляется новое значение бюджета*

II) Период активности процесса
   Планировщик отправляет к диску все запросы, пока их сумма не превышает бюджет.

III) Завершение периода активности
   1) Корректируются метки времени в WF^2Q+
   2) *Корректируется значение идеального битрейта*

Рисунок 6

Этапы, выделенные курсивом, определяются алгоритмом BFQ или MBFQ, и именно эта часть планировщика была изменена в рамках MBFQ.

9.4. Алгоритм

Алгоритм MBFQ вычисляет количество бюджета, который нужно дать процессу. Пояснения всех формул, используемых в алгоритме, могут быть найдены в пункте 9.5.

Для алгоритма необходимо хранить следующие структуры:

1) История запросов $\{\boldsymbol{s_k}, \boldsymbol{a_k}, \boldsymbol{f_k}\}$ – размер запроса, его время прихода и расчетное время окончания исполнения. Хранится для всех запросов, пришедших в течение прошлого периода T.

2) История бюджетов $\{\boldsymbol{B_i}, \boldsymbol{t_i}\}$ – бюджет, а также время, когда бюджет был изменен (т.е. когда процесс был выбран алгоритмом).

Начальные данные алгоритма:

- Веса процессов W_i по умолчанию равны $\frac{1}{N}$, где N – количество процессов. Веса также можно задавать вручную каждому процессу.
- Идеальные битрейты $b_i^{*,W}$ равны весу W_i.
- Бюджеты B_i равны $\frac{B_{max}}{2}$

Последнее требование не принципиально, т.к. для всех периодов активности, кроме первого, бюджет пересчитывается заново. Начальное значение бюджета используется только в первом периоде активности, т.к. этот момент у процесса может не быть репрезентативной истории запросов.

Шаги алгоритма для расчета бюджета i-го процесса:

1) Из истории запросов удаляются запросы, которые пришли раньше, чем $t_0 - T$, где t_0 — текущий момент времени. Т.е. удаляются запросы, для которых выполнено

$$a_k < t_0 - T$$

2) По запросам, хранящимся в истории запросов, вычисляется средний размер запросов $\overline{s_i}$ и средний интервал между запросами $\overline{t_i}$, как описано в пункте 9.5.6.

3) Из истории изменения бюджетов вычисляется $T_{inactive}$ по формуле (16).

4) Из размеров запросов в очереди процесса, а также $\overline{t_i}$ и $\overline{s_i}$ вычисляется сумма S_i размеров запросов, которые надо исполнить. S_i вычисляется по формулам (7) и (8).

5) Из S_i вычисляется предварительное значение идеального битрейта b_i^* по формуле (4)

6) Вычисляем идеальный битрейт с учетом веса $b_i^{*,W}$ по формуле (9).

7) Корректируем сумму идеальных битрейтов по всем приложениям (формула (10)). Если сумма битрейтов стала больше 1, то для всех процессов, для которых справедливо $b_i^{*,W} > W_i$, выполняем $b_i^{*,W} := W_i$.

8) По формуле (15) получаем значение нового бюджета B_{new}^{i} для i-го процесса. Если $B_{new}^{i} > B_{max}$, то $B_{new}^{i} \coloneqq B_{max}$.

После окончания периода активности:

1) Если приложение использовало не весь свой бюджет, то необходимо скорректировать идеальный битрейт следующим образом:

$$\Delta b_i^{*,W} = \frac{\Delta B_i}{(T_{active} + T_{inactive})\, b_{agg}}$$

где ΔB_i – оставшийся бюджет.

2) Если запрос синхронный, то планировщик ждет в течение времени T_{wait}, не придет ли следующий синхронный запрос. Если запрос приходит, то бюджет процесса увеличивается на $\Delta B = s_i$, но не должен превышать B_{max} (s_i – размер пришедшего запроса).

9.5. Математическое обоснование

Покажем, почему данный алгоритм позволяет добиться поставленных в пункте 9.1 целей.

9.5.1. Основные понятия и обозначения

Введем следующие обозначения:

W_i – вес i-го приложения, $W_i \in [0,1]$

b_i — битрейт i-го приложения

b_i' — битрейт i-го приложения после корректировки

b_i^* — идеальный битрейт

$b_i^{*,W}$ — идеальный битрейт с учетом веса

b_{agg} — максимальная пропускная способность диска

T — интервал времени, в течение которого собирается история запросов

B_i – бюджет i-го процесса

Поясним, что есть интервал времени T. Это интервал времени, в течение которого собирается история запросов процессов. На основе истории впоследствии предсказывается активность процесса в будущем, поэтому период T должен быть достаточно большим, чтобы история запросов за этот период была репрезентативной. В данной работе за T принимается значение $\frac{NB_{max}}{b_{agg}}$, где N – среднее число процессов в системе, предпринимающих дисковую активность, т.е. время, за которое N процессов израсходуют свои бюджеты.

Пересчет битрейта и бюджета происходит перед началом следующего периода активности. Битрейт считается за период активности T_{active}, а также за предыдущее время «простоя» приложения $T_{inactive}$ (т.е. пока оно было неактивным). На следующий период активности нужно выбрать такой бюджет, чтобы средний битрейт за $T_{active} + T_{inactive}$ соответствовал идеальному. За период активности будет прочитано или записано B_i байт (т.е. количество, равное бюджету), поэтому битрейт определяется следующим выражением:

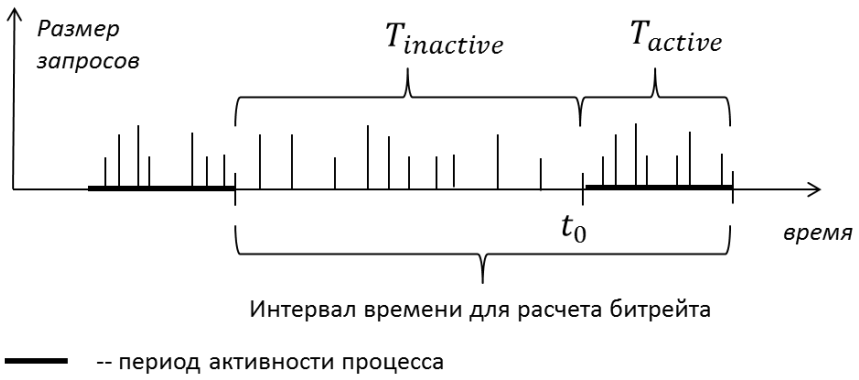


Рисунок 7

$$b_i = \frac{B_i}{(T_{active} + T_{inactive})\ b_{agg}} \qquad (3)$$

Отметим, что под битрейтом будем понимать всегда нормализованный битрейт, т.е. разделенный на максимальную пропускную способность b_{agg}:

9.5.2. Оценка идеального битрейта

Одна из целей планировщика – скорректировать битрейт так, чтобы он совпадал с идеальным битрейтом. Под идеальным битрейтом в момент времени t_0 будем понимать битрейт, который нужен процессу, чтобы исполнить все запросы из очереди, а также все запросы, которые придут во время периода активности (сумму размеров таких запросов обозначим S_i).

Итак, формула для идеального битрейта имеет следующий вид:

$$b_i^* = \frac{S_i}{T_{active} + T_{inactive}} \qquad (4)$$

Однако если какие-то из этих запросов будут исполняться долго (дольше, чем период активности), то их исполнение имеет смысл отложить до следующего периода активности. Поэтому будем считать сумму S_i только по тем запросам, у которых время завершения не позже времени окончания периода активности:

$$f_j^i = a_j^i + \frac{s_j^i}{b_{agg}} \leq t_0 + \frac{B_i}{b_{agg}} \qquad (5)$$

Здесь использованы следующие обозначения:

s_j^i – размер j-го запроса

a_j^i – время прихода j-го запроса

f_j^i – приблизительное время завершения j-го запроса

t_0 – текущий момент времени.

Поясним левую часть формулы:

$$f_j^i = a_j^i + \frac{s_j^i}{b_{agg}} \quad (6)$$

Во время периода активности i-го процесса исполняется запросы только этого процесса. Поэтому битрейт, с которым исполняются эти запросы, равен максимальному битрейту b_{agg}. Тогда $\frac{s_j^i}{b_{agg}}$ есть время исполнения j-го запроса в период активности. Таким образом, $a_j^i + \frac{s_j^i}{b_{agg}}$ равно приблизительному времени завершения j-го запроса.

Аналогично, $\frac{B_i}{b_{agg}}$ – время исполнения B_i байт, т.е. длительность периода активности. Отметим, что битрейты пересчитываются непосредственно перед тем, как процесс станет активным, и под t_0 подразумевается именно начало периода активности.

Итак, идеальным битрейтом считаем тот, который позволяет выполнить за следующий период активности все запросы, находящиеся в очереди, для которых справедливо $f_k^i \leq t_0 + \frac{B_i}{b_{agg}}$, а также запросы, которые придут во время следующего периода активности, для которых также справедливо $f_k^i \leq t_0 + \frac{B_i}{b_{agg}}$, где t_0 – текущий момент времени, f_k^i — время завершения -го запроса i-го процесса.

Подсчитаем количество запросов, которые придут в течение промежутка времени $\frac{B_i}{b_{agg}}$. Для них справедливо:

$$f_k^i = a_k^i + \frac{s_k^i}{b_{agg}} \leq t_0 + \frac{B_i}{b_{agg}}$$

где a_k^i и s_k^i— время прихода и размер k-го запроса соответственно.

Пусть запросы во время периода активности приходят через промежутки времени $\bar{t}_i$. Обозначим средний размер запроса $\bar{s_i}$. Обозначим N_r^i – количество запросов, которые придут за $\frac{B_i}{b_{agg}}$. Учитывая, что в период активности момент прихода запроса и момент начала его исполнения практически не отличаются, справедливо следующее:

$$t_0 + N_r^i \bar{t_i} + \frac{\bar{s_i}}{b_{agg}} \leq t_0 + \frac{B_i}{b_{agg}}$$

$$N_r^i \bar{t_i} + \frac{\bar{s_i}}{b_{agg}} \leq \frac{B_i}{b_{agg}}$$

$$N_r^i \le \left(\frac{B_i}{b_{agg}} - \frac{\overline{s_\iota}}{b_{agg}}\right)\frac{1}{\overline{t_\iota}} = \frac{B_i - \overline{s_\iota}}{\overline{t_\iota}\, b_{agg}}$$

Обозначим сумму размеров запросов, находящихся в очереди, для которых справедливо $f_k^i \le t_0 + \frac{B_i}{b_{agg}}$, через $\sum_k s_k^{0,i}$. Итак, подсчитаем, сколько байт надо будет прочитать или записать в следующем периоде активности (обозначим эту величину S_i).

$$S_i = \sum_k s_k^{0,i} + N_r^i \overline{s_\iota} \le \sum_k s_k^{0,i} + \frac{B_i - \overline{s_\iota}}{\overline{t_\iota}\, b_{agg}}\overline{s_\iota} \quad (7)$$

Заметим, что в случае $B_i < \overline{s_\iota}$ новые запросы не успеют прийти во время периода активности. В этом случае S_i считаем без последнего слагаемого:

$$S_i = \sum_k s_k^{0,i} \quad (8)$$

При идеальном битрейте в период активности процесс должен успеть исполнить запросы из очереди и запросы, которые придут за период активности. Идеальный битрейт, напомним, считается за период активности и предыдущий период неактивности приложения. Тогда идеальный битрейт имеет следующий вид:

$$b_i^* = \frac{S_i}{(T_{active} + T_{inactive})b_{agg}} = \frac{1}{(T_{active} + T_{inactive})b_{agg}}\left(\sum_k s_k^{0,i} + \frac{B_i - \overline{s_\iota}}{\overline{t_\iota}\, b_{agg}}\overline{s_\iota}\right)$$

Однако идеальный битрейт может оказаться чересчур большим. Если дать такой битрейт процессу, это может существенно помешать дисковой активности других процессов и нарушить их гарантии. Поэтому необходимо скорректировать идеальный битрейт с учетом веса процесса.

Идеальный битрейт с учетом веса b_i^{*W} рассчитывается следующим образом:

$$b_i^{*W} = \begin{cases} b_i^*, & b_i^* \le W_i \\ b_i^*, & b_i^* > W_i, \quad 1 - \sum_{j,j\neq i} b_j^{*W} \ge b_i^* \\ W_i, & b_i^* > W_i, \quad 1 - \sum_{j,j\neq i} b_j^{*W} < b_i^* \end{cases} \quad (9)$$

Поясним данную формулу. Если процесс хочет иметь битрейт меньше, чем вес, то планировщик обязан дать ему такой битрейт, т.к. битрейт W_i гарантируется планировщиком. Если процесс хочет получить больше, чем W_i, то нужно учесть, позволяют ли битрейты других процессов дать такой битрейт. Для этого смотрим сумму идеальных битрейтов всех остальных процессов. Если ее отличие от единицы (т.е. от максимума) больше, чем i-й процесс желает получить, то такой битрейт можно предоставить процессу.

Если же сумма битрейтов остальных процессов не позволяет дать i-му процессу такой битрейт, то планировщик дает ему гарантированный битрейт W_i.

Далее корректируем сумму $\sum_j b_j^{*W}$ с учетом нового идеального битрейта:

$$\sum_j b_j^{*W} \mathrel{+}= b_{i,new}^{*W} - b_{i,old}^{*W} \qquad (10)$$

Отметим, что сумма битрейтов $\sum_j b_j^{*W}$ может превышать единицу. Это происходит при следующем сценарии: одному из процессов понадобился большой битрейт, превышающий его вес. Сумма битрейтов других процессов позволяла дать этому процессу такой битрейт, т.е. сумма битрейтов не превысит единицу. Однако следующему активному процесс также понадобилось увеличить битрейт до величины, не превышающей его вес. В соответствии с гарантиями алгоритм не может не дать ему такой битрейт. Однако теперь сумма битрейтов превышает единицу. Для того, чтобы сделать битрейт равным 1, урежем битрейты тем процессам, у которых битрейт превышает вес:

$$\forall i: b_i^{*W} > W_i;\ b_i^{*W} \coloneqq W_i$$

По свойству 2 сумма битрейтов станет равна единице.

9.5.3. Расчет нового бюджета

Теперь вычислим, какой бюджет нужно дать процессу, чтобы его битрейт изменился на нужную величину. Мы хотим добиться, чтобы новый битрейт совпадал с идеальным: $b_i' = b_i^{*W}$. Для этого новый бюджет должен удовлетворять следующему условию:

$$b_i^{*W} = \frac{B_{new}^i}{(T_{active} + T_{inactive})\ b_{agg}} \qquad (11)$$

Отметим, что T_{active} и b_i^{*W} также зависят от B_{new}^i:

$$T_{active} = \frac{B_{new}^i}{b_{agg}} \qquad (12)$$

$$b_i^{*W}(B_{new}^i) = \frac{B_{new}^i}{\left(\frac{B_{new}^i}{b_{agg}} + T_{inactive}\right) b_{agg}}$$

Из формулы (9) следует, что b_i^{*W} может принимать два значения: b_i^* и W_i. Рассмотрим оба случая.

Случай 1

$$b_i^{*W} = W_i$$

$$W_i = \frac{B_{new}^i}{\left(\frac{B_{new}^i}{b_{agg}} + T_{inactive}\right) b_{agg}}$$

$$B_{new}^{i} = \left(\frac{B_{new}^{i}}{b_{agg}} + T_{inactive}\right) b_{agg} W_i = B_{new}^{i} W_i + T_{inactive} b_{agg} W_i$$

$$B_{new}^{i} = \frac{T_{inactive} b_{agg} W_i}{1 - W_i} \qquad (13)$$

Случай 2

$$b_i^{*,W} = b_i^* = \frac{1}{(T_{active} + T_{inactive}) b_{agg}} \left(\sum_k s_k^{0,i} + \frac{B_i - \overline{s_\iota}}{\overline{t_\iota}\, b_{agg}} \overline{s_\iota} \right)$$

$$\frac{1}{(T_{active} + T_{inactive}) b_{agg}} \left(\sum_k s_k^{0,i} + \frac{B_{new}^{i} - \overline{s_\iota}}{\overline{t_\iota}\, b_{agg}} \overline{s_\iota} \right) = \frac{B_{new}^{i}}{(T_{active} + T_{inactive})\, b_{agg}}$$

$$B_{new}^{i} = \sum_k s_k^{0,i} + \frac{B_{new}^{i} - \overline{s_\iota}}{\overline{t_\iota}\, b_{agg}} \overline{s_\iota} = \sum_k s_k^{0,i} + \frac{B_{new}^{i}}{\overline{t_\iota}\, b_{agg}} \overline{s_\iota} - \frac{\overline{s_\iota}^2}{\overline{t_\iota}\, b_{agg}}$$

$$B_{new}^{i} = \frac{\sum_k s_k^{0,i} - \frac{\overline{s_\iota}^2}{\overline{t_\iota}\, b_{agg}}}{1 - \frac{\overline{s_\iota}}{\overline{t_\iota}\, b_{agg}}} = \frac{\overline{t_\iota}\, b_{agg} \sum_k s_k^{0,i} - \overline{s_\iota}^2}{\overline{t_\iota}\, b_{agg} - \overline{s_\iota}} \qquad (14)$$

Итак, из (9), (13) и (14) получаем окончательную формулу для B_{new}^{i}:

$$B_{new}^{i} = \begin{cases} \dfrac{\overline{t_\iota}\, b_{agg} \sum_k s_k^{0,i} - \overline{s_\iota}^2}{\overline{t_\iota}\, b_{agg} - \overline{s_\iota}}, & b_i^* \le W_i \\ \dfrac{\overline{t_\iota}\, b_{agg} \sum_k s_k^{0,i} - \overline{s_\iota}^2}{\overline{t_\iota}\, b_{agg} - \overline{s_\iota}}, & b_i^* > W_i, \quad 1 - \sum_{j, j \neq i} b_j^{*W} \ge b_i^* \\ \dfrac{T_{inactive} b_{agg} W_i}{1 - W_i}, & b_i^* > W_i, \quad 1 - \sum_{j, j \neq i} b_j^{*W} < b_i^* \end{cases} \qquad (15)$$

Отметим, что если величина B_{new}^{i} меньше, чем размер следующего запроса в очереди, то имеет смысл увеличить бюджет хотя бы до размера следующего запроса. Также отметим, что если вычисленный из идеального битрейта бюджет превышает максимум, то процессу присваивается максимальный бюджет.

$T_{inactive}$ вычисляется следующим образом: в истории изменения бюджетов хранятся метки времени, соответствующие моментам изменения бюджетов процесса. Берется метка времени, соответствующая предыдущему изменению бюджета или началу периода активности (назовем ее $t_{previous}$). Продолжительность предыдущего периода активности определяется формулой

$$T_{active}^{previous} = \frac{B_{previous}}{b_{agg}}$$

$T_{inactive}$ есть разница между текущим моментом времени и окончанием предыдущего периода активности и определяется следующим выражением:

$$T_{inactive} = t_0 - \left(t_{previous} + \frac{B_{previous}}{b_{agg}} \right) \quad (16)$$

Заметим, что при расчете нового бюджета нельзя рассчитать значение B_{new}^{i} напрямую, без пересчета в битрейты (например, полагая $B_{new}^{i} = S_i$). При пересчете в битрейты контролируется следующие моменты:

- Процесс не получит слишком большой битрейт, который повлиял бы на дисковую активность других процессов.
- Процесс получает гарантированный битрейт W_i, если это ему нужно. WF^{2}Q+ контролирует, что процесс в среднем получит гарантированную долю битрейта на больших промежутках времени. MBFQ же контролирует, что процесс получает гарантированный битрейт в каждый период активности.

9.5.4. Основные свойства

Рассмотрим некоторые свойства полученного результата.

Свойство 1

Битрейт процесса b_i зависит от T_{active} следующим образом:

$$b_i = \frac{T_{active}^{i}}{T_{inactive}^{i} + T_{active}^{i}}$$

Доказательство. В самом деле, это следует из формул (11) и (12).

Свойство 2

В любой момент времени сумма всех идеальных битрейтов процессов с учетом веса не превышает 1:

$$\sum_j b_j^{*W} \leq 1$$

■ Доказательство. Обозначим:

$$S = \sum_j b_j^{*W}$$

Также обозначим сумму идеальных бюджетов остальных процессов $\sum_{j,j\neq i} b_j^{*W}$.

В начале работы планировщика битрейты всех процессов полагаются равными весу.

$$S = \sum_j W_i = 1$$

Далее рассмотрим, момент времени t_0, когда MBFQ пересчитывает бюджет для i-го процесса.

Из формулы (9) следует, что возможны три варианта:

1) $b_i^* \le W_i \Rightarrow\ b_i^{*,W} \le W_i$

2) $b_i^* > W_i;\ \sum_{j,j\neq i} b_j^{*W} + b_i^* \le 1 \Rightarrow S \le 1$

3) $b_i^* > W_i; \sum_{j,j\neq i} b_j^{*W} + b_i^* > 1 \Rightarrow\ b_i^{*,W} \le W_i$

Итак, либо $S \le 1$, либо $b_i^{*,W} \le W_i$.

Если после изменения идеального битрейта i-го процесса $S > 1$, то в ходе шага 7 алгоритма все процессы, битрейт которых превышает вес, получают битрейт, равный весу. Получаем:

$$\forall i\ b_i^{*,W} \le W_i$$

$$\sum_j b_j^{*W} \le \sum_j W_i = 1 \blacksquare$$

Свойство 3

MBFQ обладает свойством честности, гарантирует процессу получение битрейта W_i и дает гарантии на задержки исполнения запроса.

■ Доказательство. Гарантии того, что процесс получит битрейт W_i, следуют из вычисления идеального битрейта (формула (9)).

MBFQ предоставляет процессам те же гарантии честности и задержки исполнения запросов, что и BFQ (формулы (1) и (2)). В самом деле, гарантии BFQ следуют из WF^2Q+ и зависят только от максимального бюджета, веса процесса и размеров запросов, но не от метода распределения бюджета между процессами. По алгоритму MBFQ бюджет процессов также не превышает заданного максимального значения B_{max}. Поэтому для MBFQ тоже справедливы гарантии, определяемые формулами (1) и (2). ■

Свойство 4

MBFQ предоставляет как минимум столько бюджета, сколько нужно процессу для того, чтобы исполнить все свои запросы, пришедшие до или во время периода активности, если это не повлияет на битрейты других процессов и этот бюджет не превышает B_{max}.

■ Доказательство. В самом деле, в пункте 4 алгоритма MBFQ процессу присваивается бюджет больше, чем сумма размеров запросов в очереди процесса. Таким образом, в период активности процессу разрешается исполнить все запросы из очереди. Алгоритм также предоставляет процессу дополнительный бюджет, для того чтобы тот исполнил запросы, пришедшие во время периода активности. Это достигается следующими способами:

1) Если запросы синхронные, то планировщик ждет в течение времени T_{wait}, не придет ли следующий синхронный запрос. Если запрос приходит, то бюджет процесса

увеличивается на $\Delta B = s_i$, но не должен превышать B_{max} (s_i есть размер пришедшего запроса).

2) Если запросы асинхронные, то время их прихода не зависит от того, исполнились ли предыдущие запросы. Поэтому из предыдущей истории запросов можно рассчитать, сколько примерно запросов придет во время периода активности процесса. В формуле (7) рассчитывается примерное количество байт, которое необходимо добавить к бюджету, чтобы исполнить эти запросы.

Таким образом, MBFQ обеспечивает выполнение всех запросов, пришедших до или во время периода активности. Тот факт, что это не повлияет на битрейты других процессов, гарантируется формулой (9). ■

Это свойство положительно влияет на суммарную пропускную способность диска, т.к. в течение долгого времени исполняются запросы от одного и того же процесса. Скорее всего, эти запросы расположены на устройстве недалеко друг от друга, и головке диска перемещается на меньшее расстояние при переходе к исполнению следующего запроса.

9.5.5. Теорема о сравнении величин задержек запросов

Теорема 1

При использовании алгоритма планирования MBFQ в условиях изменения дисковой активности величина задержки запроса между моментом его прихода и моментом отправки к диску меньше, чем при использовании BFQ.

■ Доказательство. Рассмотрим случаи увеличения и уменьшения дисковой активности процесса.

1) Увеличение дисковой активности.

Обозначим размер j-го запроса в очереди i-го процесса как s_i^j.

Рассмотрим момент начала периода активности процесса. Обозначим период времени, в течение которого запрос s_i^j уже ждет в очереди, как $d_0^{i,j}$.

Подсчитаем величину задержки запроса, внесенную следующим периодом активности, в случае использования BFQ. Обозначим эту величину $d^{i,j}$.

Примечание. В статье [1], где впервые был описан BFQ, утверждается, что максимальная задержка не превышает максимума, определяемого формулой (2):

$$d_{i,max} \equiv max_j d_i^j \leq \frac{B_{i,max} - min_j L_i^j}{W_i b_{agg}} + \frac{B_{max} + L_{max}}{b_{agg}} \leq \frac{B_{i,max}}{W_i b_{agg}} + \frac{B_{max} + L_{max}}{b_{agg}}$$

Однако с учетом того, что W_i может принимать значения порядка $\frac{1}{10}$ для не сильно загруженных систем и порядка $\frac{1}{100}$ для серверов, $max_j d_i^j$ является очень грубой оценкой , поэтому мы оценим $max_j d_i^j$ более точно в зависимости от положения запроса в очереди.

Обозначим величину бюджета, присвоенного процессу после предыдущего периода активности, как B_i. Под увеличением дисковой активности понимаем следующее:

$$\sum_{k=1,\overline{N_i}} s_i^k > B_i \quad (17)$$

где N_i – размер очереди запросов.

Определим j_0 следующим образом:

$$\sum_{k<j_0} s_i^k \le B_i \; ; \; \forall n \ge j_0 \quad \sum_{k \le n} s_i^k > B_i$$

Такое j найдется, т.к. выполнено условие (17).

Для тех запросов, которые успеют исполниться за период активности, т.е. для всех запросов с индексом $j < j_0$, справедливо следующее:

$$\max d^{i,j} = d_0^{i,j} + \frac{B_{i,max} + L_{max}}{b_{agg}} \quad (18)$$

Поясним формулу (18). Запросу нужно дождаться, пока исполняться запросы, находящиеся перед ним в очереди, т.е. не более чем $\frac{B_{i,max}}{b_{agg}}$. Слагаемое $\frac{L_{max}}{b_{agg}}$ возникает из-за не непрерывности вычисления параметра virtual_time.

Теперь подсчитаем задержку для тех запросов, которые не успеют исполниться за текущий период активности (т.е. с индексом $j \ge j_0$). Таким запросам нужно будет дождаться следующего периода активности процесса, во время которого они, возможно, будут исполнены. Если они снова не поместятся в бюджет, то невыполненным запросам придется снова ждать следующего периода активности и т.д. Для запросов, которые исполнятся за следующий период активности, справедливо:

$$\max d^{i,j} = d_0^{i,j} + \frac{B_{i,max} + L_{max}}{b_{agg}} + B_{max} * n + \frac{B_{i,max} + L_{max}}{b_{agg}},$$

где n – количество процессов, которые будут выбраны активными между периодами активности рассматриваемого процесса. Т.к. неизвестно, через сколько периодов активности исполнится конкретный запрос, в общем случае формула для максимальной задержки выглядит следующим образом:

$$\max d^{i,j} = d_0^{i,j} + \frac{B_{i,max} + L_{max}}{b_{agg}} + \left(B_{max} * n_{m_j} + \frac{B_{i,max} + L_{max}}{b_{agg}} \right) * m_j,$$

где m_j – количество периодов активности i-го процесса, через которое исполнится j-й запрос. Отметим, что эта оценка не может превышать оценку, определяемую формулой (2).

Оценим m для запроса s_i^j. За рассматриваемый период активности не успели исполниться запросы с индексом j таким, что:

$$\sum_{k\le j} s_i^k > B_i$$

За каждый последующий период активности бюджет увеличивается на величину B_{inc}, но не превосходит максимальную величину бюджета. Поэтому m_j есть минимальное целое число, для которого выполняется следующее:

$$\sum_{k\le j} s_i^k - B_i \le \min(B_i + B_{inc}, B_{max}) + \min(B_i + 2B_{inc}, B_{max}) + \dots + \min(B_i + m_j\, B_{inc}, B_{max})$$

Заметим, что $B_i + mB_{inc} \le B_{max} \leftrightarrow m \le \frac{B_{max}-B_i}{B_{inc}}$

$$\sum_{k\le j} s_i^k - B_i \le (B_i + B_{inc}) + (B_i + 2B_{inc}) + \dots + (B_i + \frac{B_{max}-B_i}{B_{inc}} B_{inc}) + B_{max}(m_j - \frac{B_{max}-B_i}{B_{inc}})$$

$$\sum_{k\le j} s_i^k - B_i \le \frac{B_{max}-B_i}{B_{inc}} B_i + B_{inc}\frac{B_{max}-B_i}{2B_{inc}}\left(\frac{B_{max}-B_i}{B_{inc}} + 1\right) + B_{max}(m_j - \frac{B_{max}-B_i}{B_{inc}})$$

$$m_j \ge \frac{1}{B_{max}}\left(\sum_{k\le j} s_i^k - B_i + \frac{(B_{max}-B_i)^2}{2B_{inc}} - \frac{B_{max}-B_i}{2}\right)$$

Итак, m_j определяется следующим образом:

$$m_j = \left\lceil \frac{1}{B_{max}}\left(\sum_{k\le j} s_i^k - B_i + \frac{(B_{max}-B_i)^2}{2B_{inc}} - \frac{B_{max}-B_i}{2}\right)\right\rceil \quad (19)$$

Итак, для BFQ задержки запросов можно оценить следующим образом:

$$\max d^{i,j} = \begin{cases} d_0^{i,j} + \frac{B_{i,max} + L_{max}}{b_{agg}}; & \forall j: \sum_{k\le j} s_i^k \le B_i \\ d_0^{i,j} + \frac{B_{i,max} + L_{max}}{b_{agg}} + \left(B_{max} * n_{m_j} + \frac{B_{i,max} + L_{max}}{b_{agg}}\right) * m_j; & \forall j: \sum_{k\le j} s_i^k > B_i \end{cases} \quad (20)$$

где m_j определяется формулой (19).

Теперь подсчитаем задержки в случае использования MBFQ. По свойству 4 процесс получает столько бюджета, сколько ему нужно для того, чтобы исполнить все свои запросы, пришедшие до или во время периода активности, если это не повлияет на битрейты других процессов и этот бюджет не превышает B_{max}.

Рассмотрим случай, когда $1-\sum_{j,j\neq i} b_j^{*W} \geq b_i^*$, т.е. битрейт, который нужно обеспечить i-му процессу не помешает другим. Все запросы, для которых j: $\sum_{k\leq j} s_i^k \leq B_{max}$, выполнятся за первый же период активности, т.е. за $d_0^{i,j} + \frac{B_{i,max}+L_{max}}{b_{agg}}$. Запросы, не поместившиеся в бюджет, исполнятся в следующий период, если поместятся в B_{max}. Количество периодов активности, через которое исполнится запрос с j: $\sum_{k\leq j} s_i^k > B_{max}$ определяется формулой (19) при $B_i = B_{max}$:

$$m_j = r\left(\frac{\sum_{k\leq j} s_i^k - B_{max}}{B_{max}}\right) \quad (21)$$

Итак, максимальные задержки в случае MBFQ определяются следующим образом:

$$\max d^{i,j} = \begin{cases} d_0^{i,j} + \frac{B_{i,max}+L_{max}}{b_{agg}}; \quad \forall j: \sum_{k\leq j} s_i^k \leq B_{max} \\ d_0^{i,j} + \frac{B_{i,max}+L_{max}}{b_{agg}} + \left(B_{max} * n_{m_j} + \frac{B_{i,max}+L_{max}}{b_{agg}}\right) * m_j; \quad \forall j: \sum_{k\leq j} s_i^k > B_{max} \end{cases} \quad (22)$$

где m_j определяется формулой (21).

При сравнении выражений (20) и (22) видно, что задержки при использовании MBFQ меньше, чем при использовании BFQ. Во-первых, запросы с индексом j: $B_i \leq \sum_{k\leq j} s_i^k \leq B_{max}$ исполняются в случае MBFQ в первом же периоде активности, в отличие от BFQ. Для запросов с j: $\sum_{k\leq j} s_i^k > B_{max}$ количество m_j периодов активности, через которое j-й запрос исполнится, меньше в случае MBFQ (формула (21)), чем в случае BFQ (формула (19)).

Случай, когда $1-\sum_{j,j\neq i} b_j^{*W} < b_i^*$, рассматривается аналогично. В этом случае в текущем периоде активности приложение, исходя из формулы (15), получает бюджет $\frac{T_{inactive} b_{agg} W_i}{1-W_i}$.

2) Уменьшение дисковой активности.

При уменьшении дисковой активности в условиях использования BFQ бюджет процесса оказывается больше, чем сумма размеров запросов в очереди, и BFQ исполняет их все за один период активности. В MBFQ процессу присваивается бюджет больший, чем сумма размеров запросов в очереди, поэтому при использовании MBFQ все запросы также исполняться за один период активности. Итак, в случае уменьшения дисковой активности, задержки запросов, вносимые текущим периодом активности, одинаковы. ■

Теорема 2

При использовании алгоритма планирования MBFQ время реакции на уменьшение дисковой активности меньше, чем при использовании BFQ.

■ Доказательство. Пусть дисковая активность процесса уменьшилась. Запросу нужно как минимум дождаться следующего периода активности, чтобы эти изменения отразились на его бюджете. Обозначим время до следующего периода активности как t_{next}.

В случае MBFQ в начале периода активности бюджет пересчитывается исходя из текущего уровня дисковой активности, поэтому время реакции на уменьшение дисковой активности есть t_{next}.

В случае BFQ бюджет будет урезан только по окончании периода активности, т.е. время реакции в случае BFQ составляет $t_{next} + \frac{S}{b_{agg}}$, где S – сумма размеров запросов в очереди. Итак, в случае MBFQ время реакции на уменьшение дисковой активности меньше, чем в случае BFQ. ■

Отметим, что пока уменьшение дисковой активности еще не отразилось на бюджете процесса, WF^2Q+ будет выбирать следующий активный процесс с учетом старого значения бюджета, т.е. может несправедливо выбрать следующий активный процесс. Поэтому меньшее время реакции на уменьшение дисковой активности способствует более справедливому распределению дискового ресурса.

9.5.6. Оценка частоты прихода и среднего размера запросов

В предыдущем пункте для подсчета идеального битрейта использовались понятия среднего размера запроса и среднего времени между запросами. Для их вычисления используется история запросов процесса за период T. Подсчитать $\overline{s_i}$ не составляет труда – точечная оценка среднего размера запроса считается как среднее арифметическое от всех запросов, пришедших за последний период T. При подсчете же $\overline{t_i}$ возникают некоторые сложности.

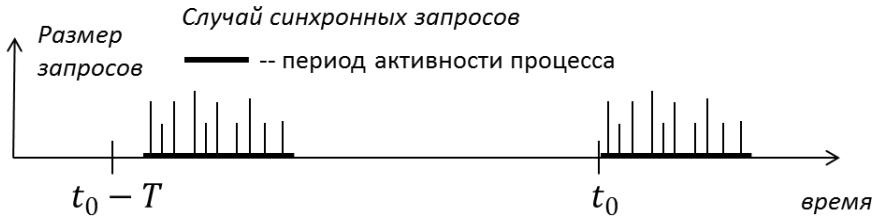
Если взять в качестве $\overline{t_i}$ среднее арифметическое от всех запросов за период T, то оценка получится необъективной. Имеет смысл рассчитывать $\overline{t_i}$ на основании типа запроса – синхронный ли он или асинхронный. Эта информация содержится в структуре IRP, в виде которой запрос приходит в файловую систему.

Рассмотрим пример типичной активности приложения в случае, если оно использует асинхронные запросы. Зачастую от таких приложений запросы приходят пачками, а не равномерно распределены по временной оси.

Подсчитав среднее время между запросами, получим среднее время между запросами в пачке, однако это не то, чего хотелось бы добиться. Хотелось бы оценить, сколько придет запросов во время периода активности, учитывая и периоды, когда приложение не посылало запросов. Для этого считаем, что приложение посылает запросы с постоянной интенсивностью $\frac{N_T}{T}$, где N_T – количество запросов, пришедших за последний период T. Тогда $\overline{t_i}$ оценим как обратную величину интенсивности:

$$\overline{t_i^{async}} = \frac{T}{N_T}$$

Теперь рассмотрим случай синхронных запросов. Здесь сложность состоит в том, что приложение посылает запрос только после исполнения предыдущего, т.е. только в период активности. Это значит, что история за прошедший период T не будет отражать активность приложения в следующий период активности. Заметим, что в период активности запросы приходят практически сразу после окончания исполнения предыдущего, т.е. среднее время между запросами можно оценить как среднее время исполнения запроса от этого приложения:



$$\overline{t_i^{sync}} = \frac{\overline{s_i}}{b_{agg}}$$

Итак, $\overline{t_i}$ рассчитывается по следующей формуле:

$$\overline{t_i} = \begin{cases} \frac{T}{N_T}, & \text{для асинхронных запросов} \\ \frac{\overline{s_i}}{b_{agg}}, & \text{для сихронных запросов} \end{cases}$$

9.5.7. Уменьшение времени ответа приложения

В данном алгоритме мы стремимся уменьшить время ответа приложения. Чтобы этого добиться, недостаточно дать приложению, с которым в данный момент работает пользователь, больше бюджета. Необходимо также, чтобы это приложение вызывалось чаще других. Один из способов добиться этого – увеличить его вес. Разумно увеличить вес ровно насколько, чтобы задержка в исполнении запросов от этого приложения была незаметна для пользователя, но BFQ выбирал не только данное приложение.

Из формулы (2) имеем, что

$$W_i \leq \frac{B_{i,max} - min_j L_i^j}{d_{i,max}\ b_{agg} - (B_{max} + L_{max})} \qquad (23)$$

$d_{i,max}$ – максимальная задержка выполнения запроса. По результатам экспериментов задержка до 100 мс незаметна для человека. Используя это значение, можно вычислить вес, который нужно присвоить процессу, чтобы оно отвечало с приемлемой скоростью. Такой подход применяется в MBFQ: приложению, которое является интерактивным, увеличивается вес до величины, рассчитанной выше. Таким образом, гарантируется, что интерактивное приложение будет испытывать задержки не более 100 миллисекунд.

Отметим, что если $d_{i,max}\ b_{agg} \leq (B_{max} + L_{max})$ или $\frac{B_{i,max} - min_j L_i^j}{d_{i,max}\ b_{agg} - (B_{max} + L_{max})} \geq 1$, то указанную задержку невозможно обеспечить. В этом случае следует увеличить вес процесса до максимально приемлемого (например, до ½).

9.6. Оценка сложности

Подсчитаем сложность алгоритма MBFQ.

Выбор следующего активного приложения осуществляется с помощью WF^2Q+, сложность которого O(log N), где N – количество процессов, имеющих запросы к диску.

Первый шаг для расчета бюджета процесса – удаление слишком старых запросов из истории. Обозначим момент времени, позже которого запросы считаются слишком старыми, как t_{old}. Запросы отсортированы по времени прихода. Этот шаг имеет оценку сложности O(1), например, храня запросы в виде хеш-таблицы, где ключ – время прихода запроса. По ключу находим запрос, который пришел в окрестности момента t_{old}, и отбрасываем часть очереди до этого запроса (т.е. еще более старые запросы.)

Далее нужно вычислить $\overline{t_i}$ и $\overline{s_i}$ из истории запросов. Этот шаг также можно сделать за O(1), если хранить сумму размеров запросов, сумму интервалов между запросами, а также количество запросов в очереди. Суммы корректируются при приходе нового запроса.

Параметры S_i, b_i^*, $b_i^{*,W}$ вычисляются за O(1).

Если сумма битрейтов стала больше 1, то для всех процессов, для которых справедливо $b_i^{*,W} \leq W_i$, нужно выполнить $b_i^{*,W} := W_i$. Это действие нельзя выполнить быстрее, чем O(N). Однако это шаг выполняет только при условии, что сумма битрейтов стала больше 1.

Шаги, выполняемые при завершении периода активности, выполняются за O(1).

Таким образом, сложность алгоритма MBFQ составляет O(N).

9.7. Приемы, заимствованные из BFQ

- Ожидание следующего синхронного запроса

Если процесс посылает синхронные запросы, то планировщику не придет новый запрос, пока не исполнится предыдущий. Поэтому когда последний из очереди процесса был отправлен диску, BFQ ждет в течение T_{wait}, не придет ли планировщику новый запрос

от этого процесса. Это очень эффективный способ увеличить битрейт процесса с синхронными запросами, несмотря на то, что диск простаивает во время ожидания T_{wait}. Исполняя больше запросов от одного процесса подряд, мы в большинстве случаев сокращаем расстояние, которое надо преодолеть головке диска. К тому же процесс не ждет, пока его снова выберут активным для исполнения следующего синхронного запроса, т.е повышается время ответа приложения.

- Правило переноса отметок времени назад.

Когда приходит новый запрос, алгоритм присваивает ему метку времени, равную моменту окончания предыдущего запроса. Это помогает в ситуации, когда запрос пришел раньше, чем алгоритм стал обрабатывать событие прихода нового запроса. Если бы алгоритм присваивал запросу текущую метку времени, гарантии задержки могли бы нарушаться.

В оригинальном BFQ запросы переупорядочиваются в очереди в соответствии с алгоритмом C-LOOK, с учетом их расположения на диске. В MBFQ это не используется, т.к. внутренний планировщик диска достаточно хорошо справляется с задачей оптимизации запросов на с учетом геометрии диска. Переупорядочивать запросы еще и в MBFQ не имеет смысла, т.к. то может мешать работе внутреннего планировщика, а также требует дополнительных временных затрат.

10. Сравнение модифицированного BFQ и оригинального BFQ на I/O тестах

Для сравнения BFQ и MBFQ были проведены тесты, подтверждающие выводы, полученные в теории. Все тесты проводились в виртуальной машине с использованием физического жесткого диска.

1) Одновременная запись несколькими процессами.

В данном тесте мы хотим показаться, что MBFQ сохраняет свойство честности, а также не страдает потерей суммарной пропускной способности.

В ходе этого теста запускались несколько процессов, которые одновременно осуществляли запись на диск, отправляя запросы разных размеров (от 32 Кб до 2 Мб), и замерялась скорость, с которой каждый из них писал на диск. Все процессы имели одинаковый вес, поэтому их битрейты в соответствии с принципом честности должны быть одинаковыми. Чтобы измерить, насколько отличались битрейты, вычислялась такая величина как среднеквадратичное отклонение битрейта (Рисунок 8). Т.к. битрейты должны быть одинаковыми, при честном распределении диска график должен быть как можно ближе к нулю. На рисунке 8 видно, при отсутствии планировщика (NOOP), график

находится далеко от нулевой оси, т.е. скорость записи распределяется нечестно в этом случае. В случае же BFQ и MBFQ график находится около нуля. На рисунке 9 та же величина, но в другом масштабе. Из этого графика видно, что MBFQ распределяет ресурсы чуть более честно, чем BFQ.

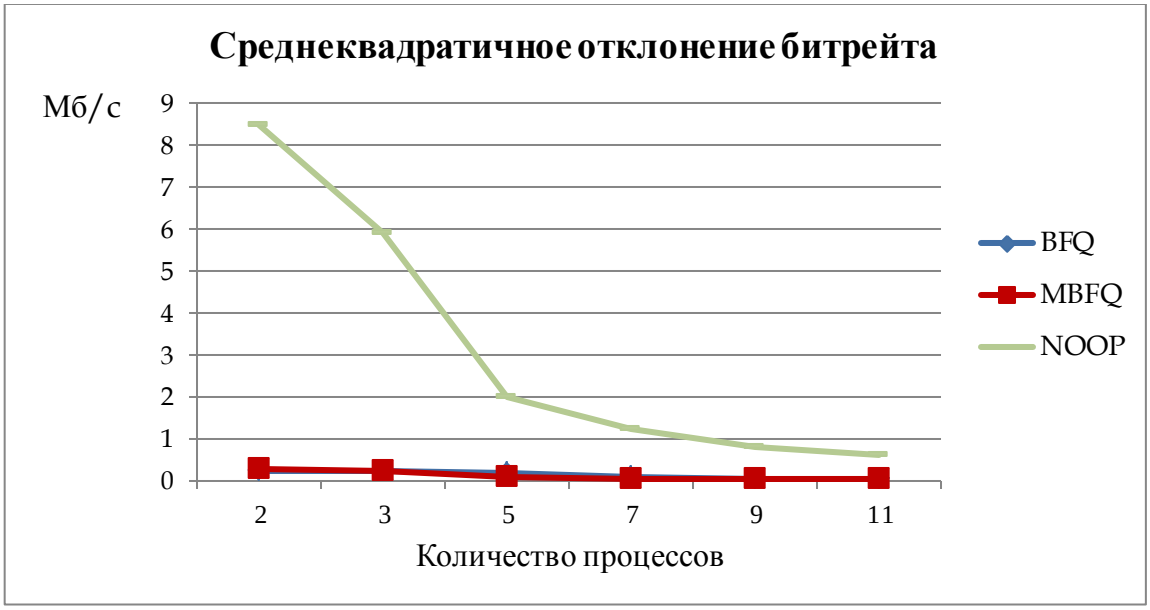


Рисунок 8

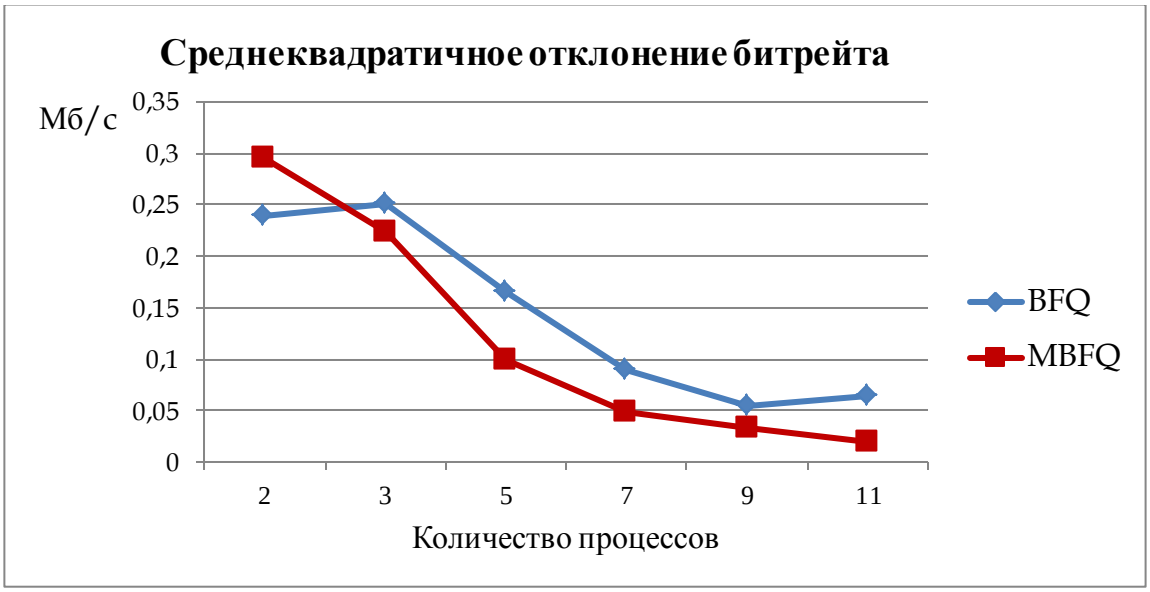


Рисунок 9

На рисунке 10 представлена суммарная пропускная способность, полученная в данном тесте. У всех обоих планировщиков, а также в случае отсутствия планировщика (NOOP) они примерно одинакова, что является хорошим результатом, т.к. многие планировщики страдают потерей суммарной пропускной способности.

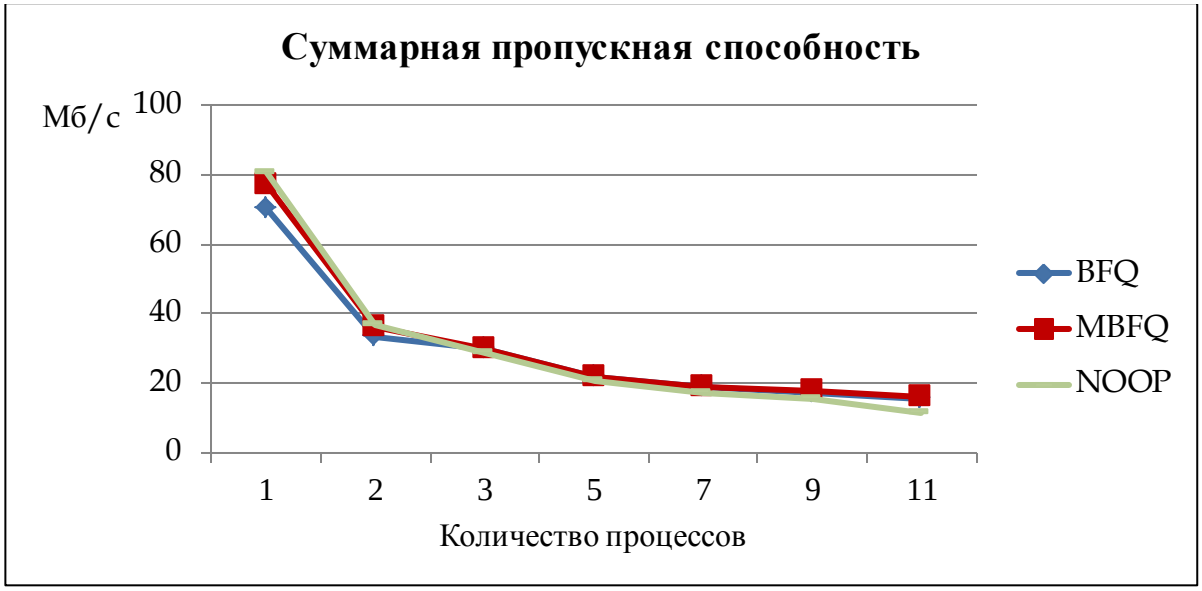


Рисунок 10

Также в этом тесте была измерена максимальная и средняя задержки между приходом запроса и его отправкой к диску (рисунки 11 и 12). На этих графиках видно, то задержка в случае MBFQ меньше, но не намного. Это связано с тем, что активность процессов постоянна, и в этом случае и не предполагалось получить большую разницу между BFQ и MBFQ.

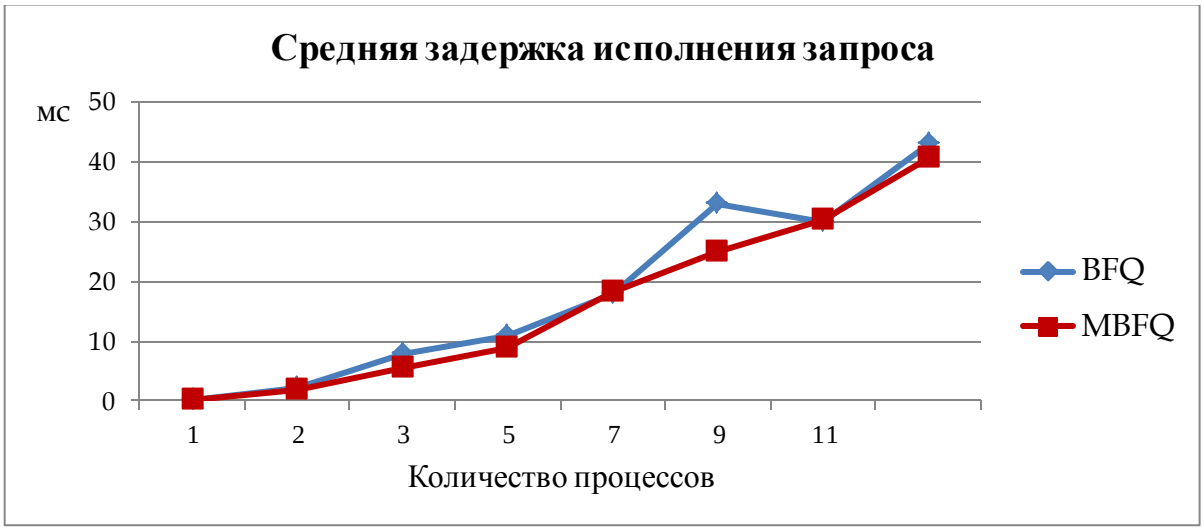


Рисунок 11

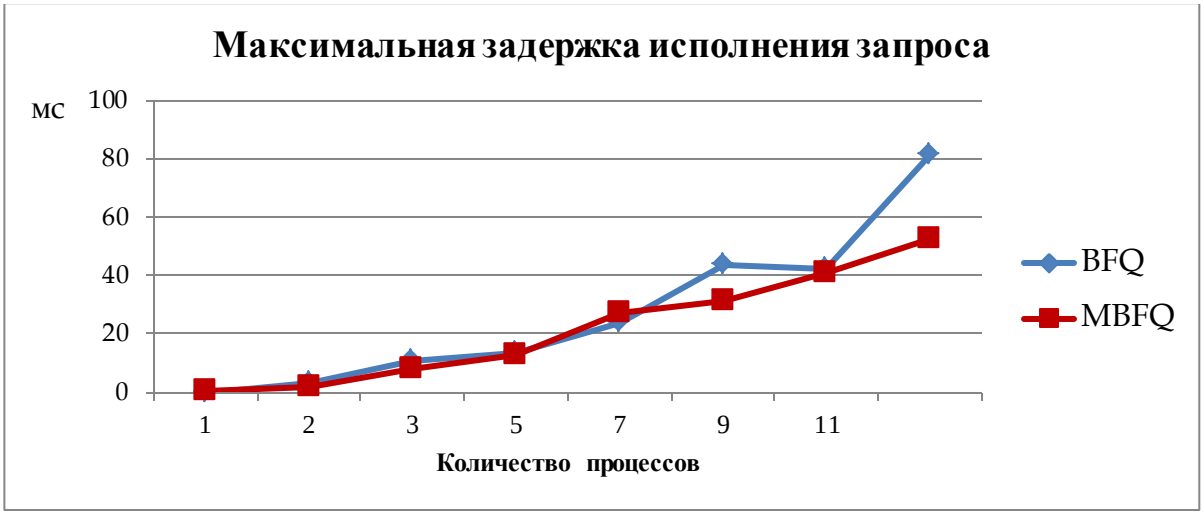


Рисунок 12

2) Время старта программ

Основной эксперимент, показывающий преимущества MBFQ – измерение времени старта программ при наличии фоновой дисковой активности (рисунок 13) и без нее (рисунок 14). Измерения проводились для таких распространенных программ как VLC, Adobe Photoshop, Microsoft Word, Visual Studio и VmWare Workstation. Стоит отметить, что время запуска VLC измерялось при открытии видео-файла размером 4 Гб, а время запуска Word – при открытии файла .doc размером 11 Мб.

Из рисунка 14 видно, что при отсутствии активности преимущества почти нет, т.к. все запросы исполняются сразу. На рисунке 13 же заметно, что при наличии дисковой активности MBFQ имеет значительное преимущество перед BFQ, давая в случае Photoshop улучшение производительности в 2 раза.

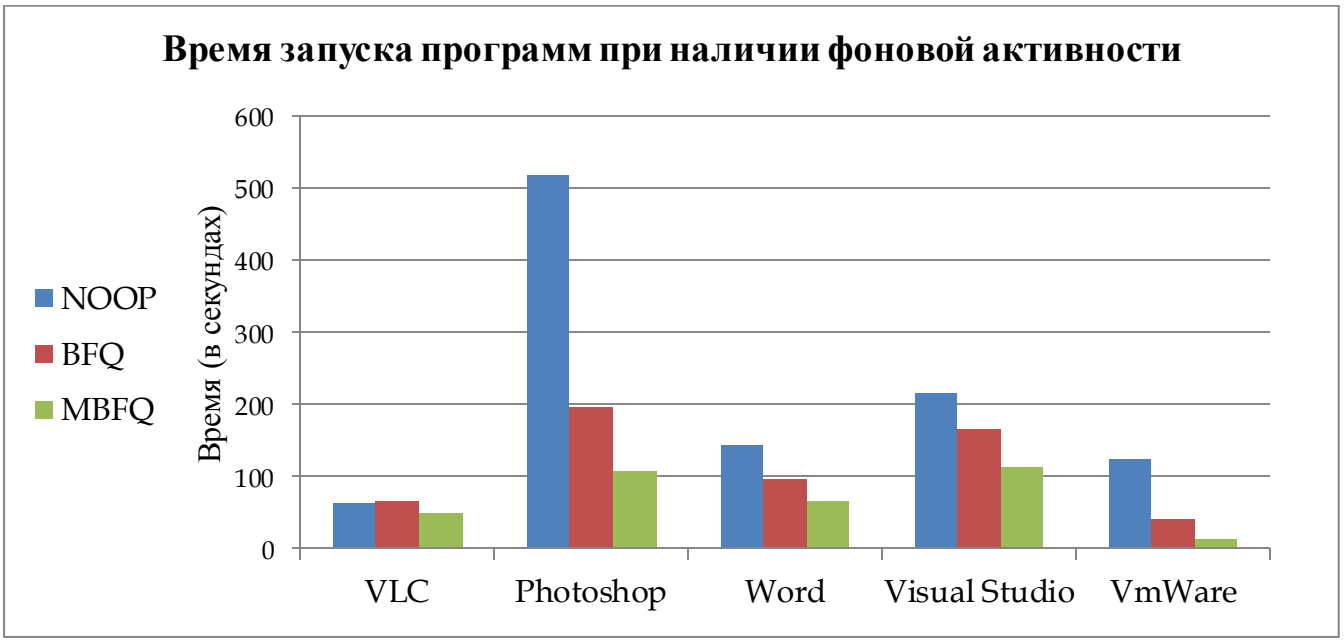


Рисунок 13

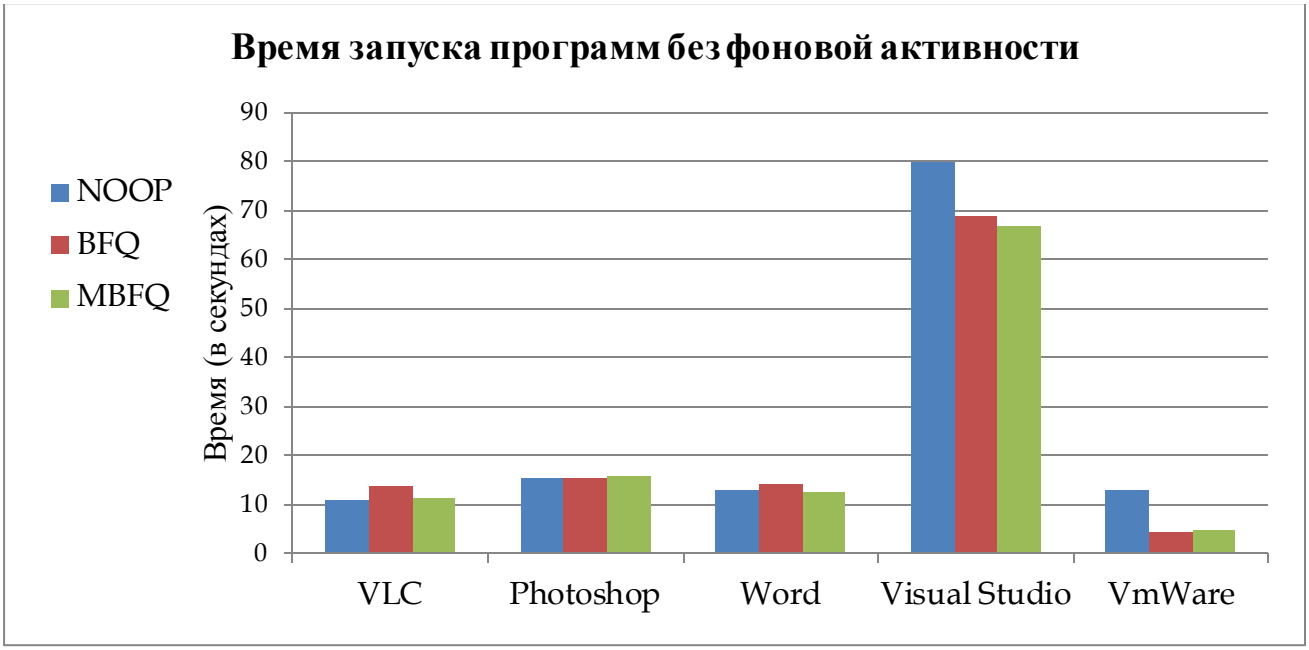


Рисунок 14

3) Медиа-сервер

Этот тест также показывает степень честности распределения ресурсов, однако не в синтетических условиях, как в первом тесте, а в реальных. В данном тесте была сделана имитация видео-сервера – на машине было запущено несколько экземпляров видеопроигрывателя VLC, каждый из который проигрывал свой видео-файл. Измерялось количество потерянных кадров в зависимости от количества одновременно проигрываемых фильмов. Чем честнее распределяется дисковый ресурс, тем более одинаковым является процент потерянных кадров для каждого экземпляра VLC и тем меньше средний процент потерянных кадров.

На графике 8 видно, что при отсутствии планировщика процент потерянных кадров растет гораздо быстрее, чем при использовании MBFQ или BFQ. На рисунке 16 представлены те же графики, что и на рисунке 15, но в другом масштабе. Из них следует, что с MBFQ процент потерянных кадров меньше, чем с BFQ.

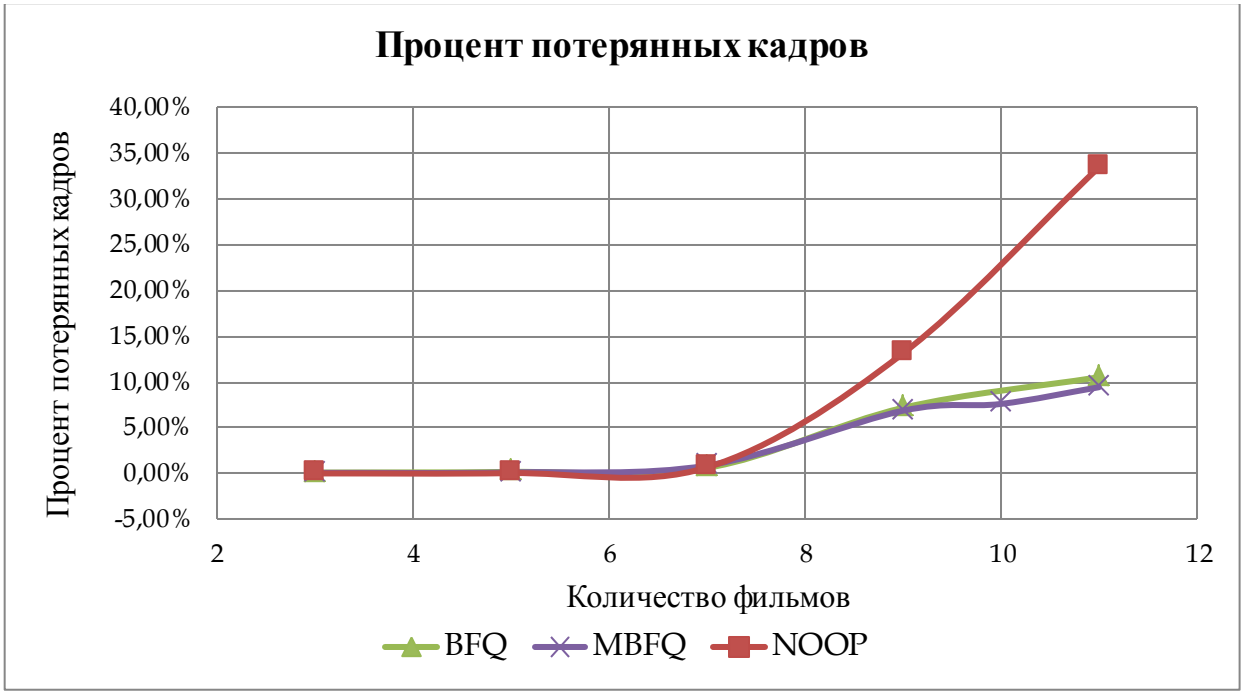


Рисунок 15

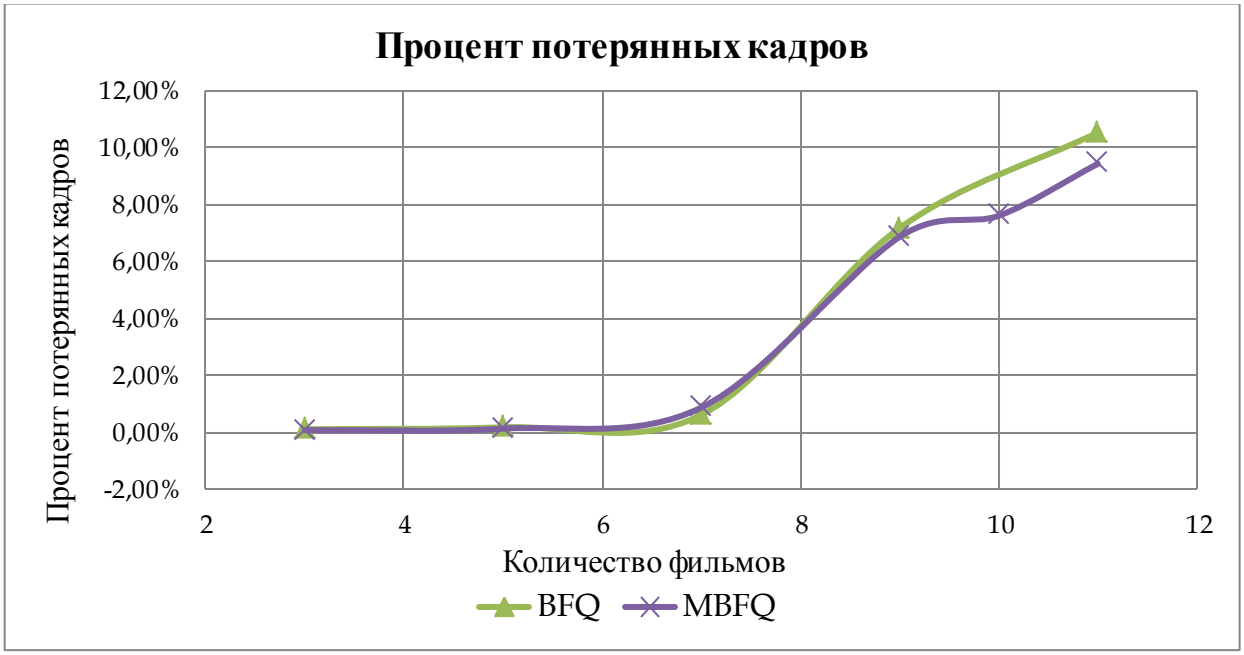


Рисунок 16

На рисунках 17 и 18 представлена дисперсия потерянных кадров. При честном распределении ресурсов график должен быть около нуля. На этих графиках видно, что MBFQ дает преимущество в честности по сравнению с BFQ.

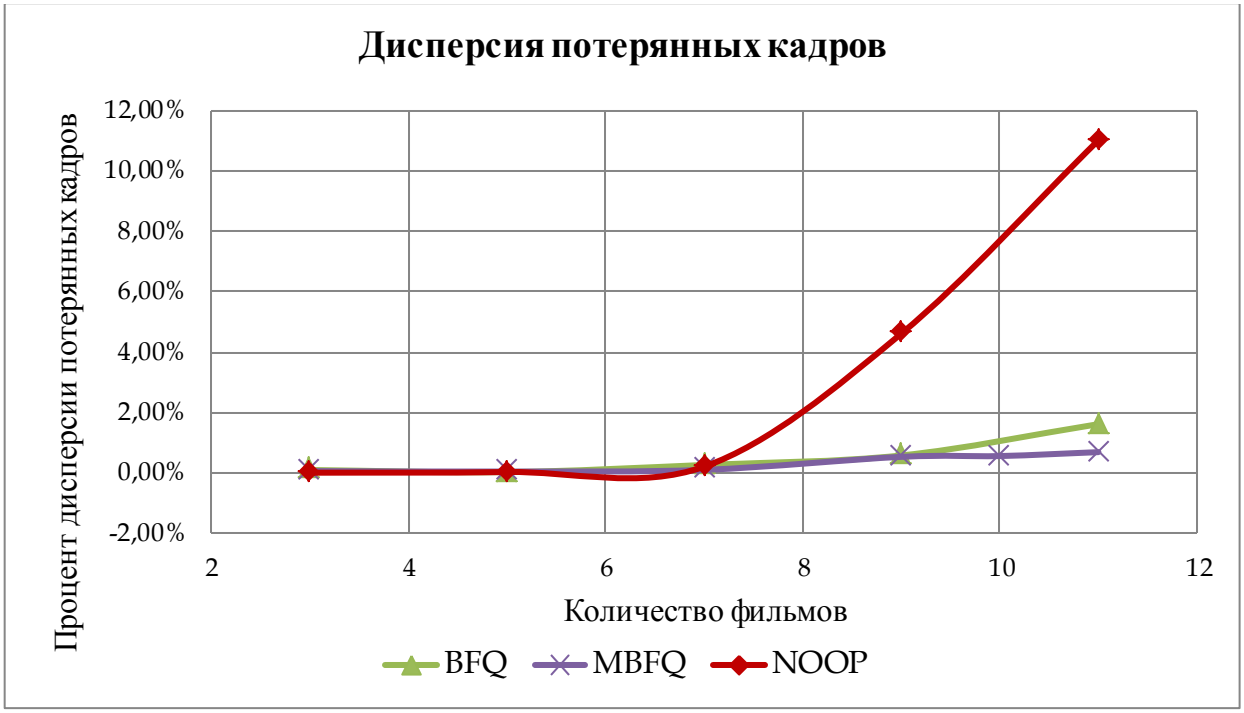


Рисунок 17

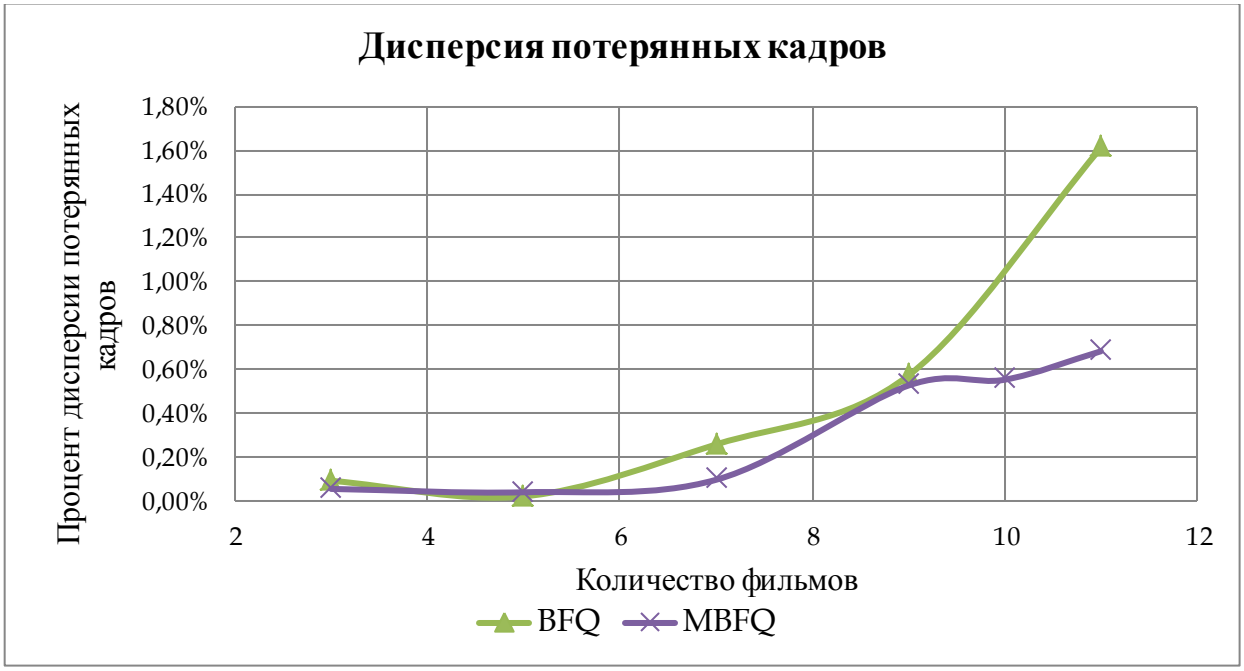


Рисунок 18

11. Дальнейшее развитие работы

Работа над планировщиком дисковой активности предоставляет большое поле для дальнейших исследований.

11.1. Учитываем, что запросы приходят в виде пакетов

В первую очередь хотелось бы учесть в алгоритме планирования интересную особенность подсистемы ввода-вывода в Windows – запросы посылаются файловой системе

в вид пакетов, и могут быть разбиты на более мелкие. Можно изучить работоспособность и эффективность следующих соображений:

- Сейчас при больших запросах (например, 1 Мб и больше) нужно давать процессу бюджет, превышающий размер запроса, т.е. минимум 1 Мб. Как уже рассчитывалось выше, при таких бюджетах задержки между приходом запроса и началом его исполнения становятся слишком большими и неприемлемыми для пользователя. Если запрос слишком большой, то планировщик может дробить его на более мелкие пакеты. Это позволит существенно сократить размер бюджета, который необходимо выделять процессу и, как следствие, сократить задержки исполнения.
- Допустим, синхронный запрос был разбит на более мелкие. Имеет смысл находить в очереди пакеты, которые относятся к одному и тому же запросу. В частности, если были исполнены все части запроса, кроме одной (по какой-либо причине – закончился бюджет, либо она отделена в очереди от других частей запроса еще какими-то запросами), то можно увеличить бюджет и выполнить эту оставшуюся часть запроса. Тогда приложение, от которого пришел этот запрос, продолжит свою деятельность сразу же. В противном случае ему придется ждать следующего периода активности.

11.2. Использование бюджета полностью

Выше уже отмечалось, что гарантии соблюдаются лучше, если приложение использует весь свой бюджет. В связи с этим предлагается улучшить алгоритм планирования следующим образом:

- Допустим, на следующий запрос в очереди не хватает бюджета, однако если увеличить бюджет на маленькую величину, то оставшегося бюджета хватит, чтобы выполнить следующий запрос. В таком случае следует увеличить бюджет и выполнить запрос, на который раньше не хватало бюджета.
- Если бюджет не израсходован полностью, и в очереди еще остались запросы (т.е. следующий запрос в очереди уже не помещается в бюджет), то можно найти в очереди запрос, подходящий по размеру, чтобы израсходовать оставшуюся часть бюджета.
- Можно заметить, что многие приложения шлют запросы одинакового размера. Поэтому можно каждому приложению присваивать бюджет кратный стандартному размеру его запросов, чтобы использовать бюджет полностью.

11.3. Учитываем внутренний планировщик диска

Еще одним направлением дальнейшей работы является изучение влияния внутреннего планировщика диска на гарантии честности и время ответа приложения. Также, возможно, потребуется внести корректировки с MBFQ с учетом того, что на диске имеется еще один

планировщик, чтобы последний по крайней мере не сводил на нет преимущества MBFQ. Например, заставить MBFQ не вынимать из очереди запросов процесса все запросы, даже если это позволяет бюджет. Вместо этого можно вынуть ограниченное число запросов, дождаться их выполнения и только потом вынимать оставшиеся. Дело в том, что если вынимать сразу все запросы из очереди, то внутренний планировщик диска может их сильно переупорядочить, нарушив тем самым гарантии. К примеру, запросы, лежащие в начале очереди, могут быть исполнены в конце очереди. Более того, если MBFQ после этого выбирает следующий активный процесс и посылает его запросы диску, то запросы от предыдущего активного процесса могут быть выполнены после выполнения запросов от текущего активного процесса, если так будет выгоднее исходя из расположения запросов на диске. В этом случае гарантии нарушаются еще больше.

11.4. Оценка суммарной пропускной способности диска

При расчетах битрейта и бюджета часто используется понятие суммарной или максимальной пропускной способности диска. Однако у каждого диска свое значение пропускной способности. Отсюда возникает вопрос, как оценить это параметр.

В BFQ применяется следующий подход: для каждого активного процесса измеряется время, в течение которого он исполнял свои запросы, и сумма размеров этих запросов. Из этих параметров и вычисляется битрейт, с которым читал или писал активный процесс. Если битрейт оказался больше, чем имеющееся значение максимального битрейта, то максимальный битрейт корректируется, причем таким образом, чтобы максимальный битрейт изменялся не скачками, а более или менее равномерно. Учитывается также, что максимальная пропускная способность диска достигается только если в течение долгого времени исполнять запросы от одного и того же процесса, особенно если процесс исполняет последовательные чтение или запись. Поэтому битрейт пересчитывается только для приложений, исполнявших запросы больше 20 миллисекунд, т.к. иначе можно получить неправдоподобные результаты. Следующий код осуществляет пересчет максимального битрейта:

```
if ((Time >= 20 ms) && (bytesProcessed / Time > MaximumBitrate))
{
    MaximumBitrate = (bytesProcessed / difference) * 1/8 + MaximumBitrate * 7/8 ;
}
```

Как уже упоминалось, при реализации BFQ и MBFQ в Windows используется другая модель отправки запросов к диску, чем в BFQ в Linux. В BFQ в Linux планировщик выбирает и отправляет на исполнение следующий запрос только когда исполнился предыдущий. В Windows, напротив, к диску отправляется сразу все запросы, которые

бюджет позволяет исполнить, не дожидаясь окончания уже отправленных. По этой причине оценить, сколько времени активный процесс исполнял свои запросы, сложнее. Необходимо будет отлавливать события окончания исполнения запросов и определять, к какому процессу и к какому периоду активности относилось их исполнение.

11.5. Исследование зависимости производительности системы от максимального бюджета

Производительность системы сильно зависит от величины максимального бюджета. Также от максимального бюджета зависят гарантии процесса, как это можно видеть из формул (1) и (2). По этой причине важно исследовать, какой диапазон максимальных бюджетов больше всего подходит для работы на пользовательской машине и обеспечивает быстрый уровень ответа большинства программ.

12. Заключение

В ходе данной работы был разработан и реализован под ОС Windows планировщик дисковой активности MBFQ. Также был реализован под ОС Windows линуксовый алгоритм планирования дисковой активности BFQ, считающийся на данный момент лучшим честным планировщиком. В данной работе было показано как теоретически, так и практически, что MBFQ дает более хорошие результаты, чем BFQ. В рамках теоретического доказательства была построена математическая модель дисковой активности, на основе которой было проведено исследование свойств разработанного алгоритма. В качестве практической части были проведены тесты, позволяющие измерить задержки, испытываемые запросами, суммарную пропускную способность диска, а также степень честности распределения битрейтов между процессами. Тесты подтвердили полученные теоретические выводы о том, что MBFQ дает преимущество перед BFQ в плане задержек между приходом запроса и его отправкой на исполнение, при этом сохраняя свойство честности. Таким образом, разработанный алгоритм планирования MBFQ решает проблемы планирования дисковой активности в ОС Windows и обладает более сильными свойствами, чем аналогичные решения для ОС Linux.

Список литературы

1. P. Valente and F. Checconi, High throughput disk scheduling with fair bandwidth distribution. Computers, IEEE Transactions on, 2010. 59(9): p. 1172-1186.
2. 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, New York, NY, USA

3. Budget Fair Queueing [Электронный ресурс] http://algogroup.unimore.it/people/paolo/disk_sched
4. BFQ Test Results [Электронный ресурс] http://retis.sssup.it/~fabio/linux/bfq/results.php
5. J. Kumar, S. M. Parekh. Generalized Processor Sharing Approach to Flow Control In Integrated Services Networks. Massachusetts Institute of Technology
6. Лав Р. Разработка ядра Linux. 2-е изд. М.: Вильямс, 2006
7. Solomon D., Russinovich M. Inside Microsoft Windows 2000. 5rd edition. Microsoft Press, 2009.
8. B. L. Worthington, G. R. Ganger, and Y. N. Patt, "Scheduling algorithms for modern disk drives," in SIGMETRICS '94: Proceedings of the SIGMETRICS conference on Measurement and modeling of computer systems. New York, NY, USA: ACM, 1994, pp. 241–251.
9. D. Jacobson, J. Wilkes. Disk scheduling algorithms based on rotational position. Concurrent Systems Project, Palo Alto, CA, 1995
10. A. L. N. Reddy, J. Wyllie, and K. B. R. Wijayaratne, "Disk scheduling in a multimedia I/O system," ACM Trans. Multimedia Comput. Commun. Appl., vol. 1, no. 1, pp. 37–59, 2005.
11. Ray-I Chang, Wei-Kuan Shih. Real-Time Disk Scheduling for Multimedia Applications with Deadline-Modification-Scan Scheme. Institute of Information Science.
12. S. Daigle and J. Strosnider, "Disk Scheduling for Multimedia Data Streams,", in Proc. of the IS&T/SPIE, 1994.
13. A. Reddy, J. Wyllie. I/O issues in a multimedia system. IBM Almaden Research Center. M. Dunn, A. Reddy. A New I/O Scheduler for Solid State Devices. Texas A&M University.
14. M. Shreedhar, G.Varghese. Efficient Fair Queueing Using Deficit Round Robin. IEEE/ACM Transactions on Networking, vol. 4, no. 3, June 1996.
15. Яремчук С. Планировщики Linux [Электронный ресурс] http://www.opennet.ru/base/sys/linux_shedulers.txt.html
16. J. C. R. Bennett and H. Zhang, "Hierarchical packet fair queueing algorithms," IEEE/ACM Transactions on Networking, vol. 5, no. 5, 1997.
17. J.Bennett, H. Zhang. Worst-case Fair Weighted Fair Queueing. School of Computer Science.

18. P. Valente, “Extending WF2Q+ to support a dynamic traffic mix,” Advanced Architectures and Algorithms for Internet Delivery and Applications, 2005. AAA-IDEA 2005. First International Workshop on, pp. 26–33, 15-15 June 2005.
19. Y. Yu, D. Shin, H. Eom, and H. Yeom, “NCQ vs I/O Scheduler: Preventing Unexpected Misbehaviors,” ACM Transaction on Storage, vol. 6, no. 1, March 2010.
20. B. Dees, “Native Command Queuing - Advanced Performance in Desktop Storage,” in Potentials. IEEE, 2005, pp. 4–7.
21. “Tagged Command Queuing,” 2010, [Электронный ресурс] http://en.wikipedia.org/wiki/Tagged Command Queuing.