Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики

Магистерская диссертация

Новая неблокирующаяся на дисковом вводе реализация системного вызова sendfile() для FreeBSD.

Работу выполнил:
Смирнов Глеб Александрович
Научный руководитель:
с.н.с., к.ф.-м.н.
Гамаюнов Денис Юрьевич

Москва
2014

Оглавление

Аннотация 2

Введение 2

1 Передача файлов и *sendfile()* **4**

1.1 Ранние реализации файловых серверов 4

1.2 Отдача файлов по HTTP 4

1.3 Взаимодействие HTTP-сервера и операционной системы 5

1.4 Появление *sendfile()* 6

2 Архитектура высокопроизводительного HTTP-сервера **8**

2.1 Проблема диспетчеризации большого числа одновременных соединений 8

2.2 Время выполнения системных вызовов 9

2.3 Избежание блокирования на дисковом вводе 10

2.3.1 Выполнение чтения с диска в тредах 10

2.3.2 POSIX Asynchronous I/O 10

2.3.3 *sendfile(2)* с запретом чтения с диска 11

2.3.4 *aio_mlock(2)* перед *sendfile(2)* 12

2.3.5 *aio_sendfile(2)* 12

3 Текущая реализация *sendfile()* в FreeBSD **14**

3.1 *read(2)* файла и *write(2)* в сокет 14

3.2 Цикл *sendfile(2)* в ядре 15

3.3 Оптимизация: вложенный цикл 16

4 Новая реализация *sendfile(2)* **18**

4.1 Асинхронный VOP-метод 18

4.1.1 *VOP_READ_ASYNC()*? 19

4.1.2 *sendfile()* работающий поверх пейджера виртуальной памяти 20

4.1.3 *VOP_GETPAGES_ASYNC()* 22

4.2 “Неготовые” данные в сокетных буферах 23

4.2.1 Сокетный буфер 23
4.2.2 Сокетный буфер с “неготовыми” данными 24
4.2.3 Работа сетевых протоколов с “неготовыми” данными 25
4.3 Новый *sendfile(2)* 26
4.3.1 *sendfile_swapin()* 26
4.3.2 Основной цикл 28
4.3.3 Завершение чтения: *sf_iodone()* 30
4.4 Оптимизация и расширение API нового *sendfile(2)* 31
4.4.1 Запрет кэширования 31
4.4.2 Приложение указывает readahead 32

5 Анализ результатов **33**
5.1 Сравнение существующей и предлагаемой реализаций в прикладной задаче 33
5.2 Сравнение существующей и предлагаемой реализаций в синтетическом бенчмарке 36
5.2.1 Трафик 37
5.2.2 Загрузка CPU 38

Заключение **39**

Литература **40**

Аннотация

Работа посвящена новой реализации системного вызова *sendfile()* в ядре операционной системы FreeBSD.

В работе рассматривается проблема отдачи статических файлов HTTP-сервером в ретроспективе, разбираются эволюция механизмов отдачи, архитектура HTTP-серверов, объясняются предпосылки для новой реализации. Детально разбираются проделанные изменения в ядре FreeBSD для создания неблокирующегося на дисковом вводе *sendfile()*. Проводится сравнение нового *sendfile()* с предыдущей реализацией в бенчмарках. Демонстрируется работа новой реализации в реальных задачах.

Введение

Согласно различным оценкам, HTTP-трафик составляет от половины до трёх четвертей трафика современного Интернета. Весь этот трафик генерируется с помощью такого класса программного обеспечения, как HTTP-сервер. Одной из частных задач HTTP-сервера является отдача клиенту статического контента, то есть заранее подготовленных файлов с локального диска. Эта, на первый взгляд, тривиальная задача решается далеко не оптимально в рамках того набора системных вызовов, что предоставляет POSIX. В связи с этим большинство современных операционных систем предоставляют системный вызов *sendfile()*, который решает эту задачу намного эффективнее. Этот системный вызов появился в 1997 году, и практически все современные HTTP-серверы отсылают статические файлы с его помощью, если его предоставляет операционная система. Однако, по мере роста производительности аппаратного обеспечения, а также роста числа HTTP-клиентов, одновременно обслуживаемых одним HTTP-сервером, выявились новые проблемы масштабируемости HTTP-серверов. Время, которое HTTP-сервер проводит в отдельном системном вызове, стало критичным при одновременной работе с десятками тысяч клиентов. И *sendfile()*, системный вызов, читающий данные с диска, оказался одним из самых длительных. Для того, чтобы обойти эту проблему, были применены различные подходы, такие как задействование асинхронного ввода/вывода по POSIX или изменение архитектуры HTTP-сервера на многопоточную. Авторы данной работы предлагают решить эту проблему в корне, то есть реализовать *sendfile()* как неблокирующийся на дисковом вводе системный вызов. Такая реализация потребует ряда нетривиальных изменений в важных внутренних подсистемах ядра FreeBSD, таких как виртуальная память, VFS и сокеты.

Постановка задачи

Цель данной работы - увеличить производительность и масштабируемость системного вызова *sendfile()* за счёт избежания блокирования вызова на дисковом вводе. Для достижения указанной цели необходимо решить ряд задач:

- В подсистеме VFS ядра операционной системы FreeBSD предоставить новый метод, который позволит получать страницы файла асихронно. Реализовать этот метод для файловой системы UFS/FFS.
- В подсистеме виртуальной памяти предоставить новый API для пейджеров, позволяющий запрашивать у пейджеров страницы асинхронно. Реализовать этот API для vnode-пейджера, то есть пейджера, работающего над файлами.
- В подсистеме сокетов реализовать новую концепцию "неготовых" данных в сокете. Такие данные занимают своё место в сокете, однако не могут быть посланы в сеть до тех пор, пока не завершится ввод с диска.
- Наконец, на основе вышеперечисленных нововведений реализовать системный вызов *sendfile()* таким образом, что вызов будет возвращаться из ядра, не дожидаясь окончания ввода данных с диска, при этом данные в сокете будут консистентны.

Глава 1

Передача файлов и *sendfile()*

1.1 Ранние реализации файловых серверов

До появления TCP/IP копирование файлов между удалёнными машинами осуществлялось с помощью протокола UUCP (UNIX to UNIX CoPy) непосредственно по физическим линиям. И хотя UUCP можно запустить поверх TCP/IP, сразу после появления Интернет стала очевидна необходимость более удобных протоколов для передачи файлов. Первым протоколом получившим действительно широкое распостранение стал FTP (File Transfer Protocol) в 1980 г [1]. Непосредственно передача файлов в этом протоколе была реализована максимально просто: стороны создавали новое TCP соединение, одна из сторон передавала файл и закрывала соединение. В более поздних версиях протокола появилась возможность передачи файла не целиком, а частями. В начале 90-х годов обрёл некоторую популярность протокол Gopher, который позволял работать с удалёнными библиотеками документов [2]. Заметим, что получение документа (или части документа) из удалённой библиотеки представляет собой, по сути, частный случай передачи файла. Параллельно с Gopher также развивался протокол HTTP, впервые документированный в 1991 году [3]. В 1996 году вышла версия 1.0 протокола HTTP [4], после чего HTTP стремительно вытеснил Gopher в течение нескольких лет. С тех пор HTTP становится самым распостранённым протоколом передачи данных в сети Интернет. Протокол FTP продолжает использоваться и по сей день, однако роль его становится всё более и более второстепенной по отношению к HTTP.

1.2 Отдача файлов по HTTP

Изначально придуманный для гипертекста, HTTP, на самом деле, подходит для передачи файлов любого типа. Действительно, если гипертекст содержит

изображения, то удобно использовать этот же протокол для передачи не только текста, но сопутствующих ему изображений. Как правило, типичная страница современного веб-сайта, предназначенного для просмотра в браузере, генерируется динамически. Однако, страница содержит множество статических элементов: изображения, flash, видео. Эти элементы хранятся на диске HTTP-сервера в виде файлов и отдаются клиентам без какой-либо обработки, немодифицированными.

Помимо сайтов предназначенных для просмотра в браузере, существуют также файловые архивы, например, репозитарии для обновления программного обеспечения. Такие сайты отдают исключительно статический контент, причём объём передаваемых данных одному клиенту, как правило, на порядки больше, чем в случае отдачи гипертекста для браузера. Отдельно стоит выделить сайты, обеспечивающие просмотр видео online. Эти сервисы являются основными генераторами трафика в современном Интернете и тоже работают по HTTP. Так, в первой половине 2014 года, трафик видео-сервиса Netflix составил 34.21% всего трафика Северной Америки, а трафик Youtube ещё 13.19% [5]. Можно смело утверждать, что статический контент образует более половины всего трафика современного Интернет.

1.3 Взаимодействие HTTP-сервера и операционной системы

Рассмотрим на низком уровне процесс передачи файла клиенту с помощью приложения HTTP-сервера. Впрочем, нижесказанное будет применимо и к FTP, и к прочим протоколам. Файл представляет собой последовательность блоков на жёстком диске, которую операционная система представляет приложению как непрерывную адресуемую последовательность байт. Таким образом, работа приложения с файлом достаточно проста, приложение может читать из файла заданное количество данных по заданному смещению в буфер памяти. Это осуществляется с помощью системного вызова *read(2)*, специфицированного POSIX [6]. Осталось отправить данные из буфера в TCP-соединение, на удалённой стороне которого ожидает данных HTTP-клиент. И здесь POSIX-совместимая операционная система предоставляет удобный потоковый интерфейс сокетов, беря сложную реализацию TCP/IP полностью на себя. Приложению же предоставляется сокет - двунаправленный неадресуемый поток байт, в который можно посылать данные с помощью системного вызова *write(2)*. Таким образом, элементарную операцию записи *nbytes* байт из уже открытого файла (дескриптор *fd*) в уже открытый сокет

(дескриптор *sd*) можно отобразить следующим очень простым циклом. [1]

```
char buf[BUFSIZE];
size_t rem, len;

rem = nbytes;
for (rem = nbytes; rem > 0; rem -= len) {
        len = min(BUFSIZE, rem);
        read(fd, buf, len);
        write(sd, buf, len);
}
```

Листинг 1.1: Элементарный цикл, отправляющий данные из файла в сокет

Примерно такой код можно найти в первых реализациях FTP-серверов. Очевидно, что приведённый код будет блокироваться на записи в сокет и не пригоден для работы с множеством клиентов одновременно из одного процесса, поэтому реальные приложения устроены значительно сложнее. Этой проблемы мы ещё коснёмся в главе 2. Сейчас подчеркнём, что независимо от сложности приложения, со стороны ядра операционной системы работа приложения всё равно будет выглядеть как чередующиеся вызовы *read(2)* и *write(2)*, в чём легко можно убедиться с помощью трассировщика системных вызовов. Также важно отметить, что большая часть процессорного времени проходит в системных вызовах, а не в коде самого приложения. Фактически отдача файла выполняется операционной системой, а приложение играет лишь управляющую функцию: открывает файлы и сокеты и перенаправляет потоки данных. Таким образом, если мы хотим оптимизировать задачу отдачи статического контента, мы должны оптимизировать ядро операционной системы, предоставляя приложениям более производительные интерфейсы работы с файлами и сокетами. Со стороны приложения манёвра для оптимизации практически нет.

1.4 Появление *sendfile()*

В цикле в листинге 1.1 буфер отсылаемых данных совершенно не модифицируется между *read(2)* и *write(2)*. Закономерно встаёт вопрос об оптимизации этого кода. Действительно, копирование данных из ядерного адресного пространства в адресное пространство процесса только для того, чтобы затем скопировать их об-

[1] Листинги в работе представляют собой максимально упрощённый псевдокод, на языке подобном C. Обработка ошибок в листингах опущена ради наглядности.

ратно - напрасная трата вычислительных ресурсов. Что будет, если предоставить системный вызов, который внутри ядра осуществит передачу немодифицированных данных из файла в сокет?

Такой системный вызов впервые появился в HP-UX 11.00 в 1997 году [7]. Новый системный вызов получил имя *sendfile(2)*. В следующем году аналогичный системный вызов появился в Linux 2.2 и FreeBSD 3.0. Одновременно появление *sendfile(2)* сразу в нескольких операционных системах свидетельствует о его высокой востребованности в то время. Сообщение в репозитории FreeBSD, комментирующее появление нового системного вызова, говорит о том, что новый вызов разработан совместно с Apache Group, авторами наиболее популярного на то время HTTP-сервера, и что новый механизм отдачи файлов в 10 раз быстрее традиционного цикла *read(2)/write(2)* [8].

Глава 2

Архитектура высокопроизводительного HTTP-сервера

2.1 Проблема диспетчеризации большого числа одновременных соединений

В 1990-е годы основным HTTP-сервером в экосистеме UNIX-подобных операционных систем был Apache 1. Его архитектура была типичной для TCP-сервера того времени: для каждого нового соединения мастер-процесс порождал потомка и потомок работал с этим соединением, в частности, посылал файлы с помощью цикла, подобного циклу из листинга 1.1. В начале 2000-х годов количество абонентов Интернет стало стремительно расти, также как и количество сервисов Интернет, потребляемых одним абонентом, что привело к росту количества одновременных TCP-соединений, обслуживаемых отдельным HTTP-сервером. Выяснилось, что, во-первых, порождать новый процесс на каждое соединение чрезвычайно неэффективно, а во-вторых, классические интерфейсы POSIX совершенно непригодны для работы с тысячами дескрипторов. Этот феномен получил имя "C10K problem" или "проблема 10 тысяч соединений" [9]. Разработчики операционных систем и HTTP-серверов стали предлагать различные подходы к решению этой проблемы. Со стороны разработчиков операционных систем были предложены новые интерфейсы для работы с большим числом дескрипторов, такие как *kqueue(2)* в FreeBSD [10] и *epoll(2)* в Linux [11]. Появились новые HTTP-серверы, внутренняя архитектура которых значительно отличается от Apache. Так, современный сервер nginx имеет фиксированное количество рабочих процессов, также называемых worker

processes, и эти процессы делят между собой поровну TCP-соединения. Как правило, число worker-ов того же порядка, что и число физических процессоров в сервере. Каждый worker представляет собой конечный автомат, работающий с соединениями поочерёдно. Переключение между соединениями управляется с помощью вышеупомянутых диспетчеров событий *kqueue(2)* или *epoll(2)*. Эти архитектурные решения позволили успешно преодолеть “C10K problem”.

2.2 Время выполнения системных вызовов

Максимальное количество соединений, которое один worker сможет обрабатывать без ухудшения качества сервиса, определяется тем, как долго длятся системные вызовы, необходимые для обслуживания очередного события, поступившего от диспетчера событий. Рассмотрим, какие системные вызовы осуществляет HTTP-сервер во время своей работы:

- Открытие/закрытие файлов: *open(2)*, *close(2)*.
- Открытие/закрытие сокетов: *accept(2)*, *close(2)*, *shutdown(2)*.
- Дисковый ввод: *read(2)*.
- Чтение из сокета/запись в сокет: *read(2)*, *write(2)*.
- Дисковый ввод и запись в сокет: *sendfile(2)*.

Как долго могут длиться эти системные вызовы? Любой ввод/вывод в сокеты может заблокироваться, если в сокете нет данных при чтении или нет свободного места при записи. Время ожидания может исчисляться секундами. Такое поведение приемлемо для архитектуры «один процесс-одно соединение», но совершенно неприемлемо для новой архитектуры. Необходимый инструмент для решения этой проблемы предоставлен в рамках POSIX. Это режим неблокирующегося ввода/вывода: *fcntl(O_NONBLOCK)*. Если сокетные дескрипторы переведены в этот режим, то они будут немедленно возвращать код ошибки *EAGAIN* вместо того, чтобы блокироваться. Таким образом, проблема с блокирующимися сокетами решена.

Системные вызовы могут блокироваться и на дисковом вводе/выводе. К сожалению, исторически так сложилось, что флаг *O_NONBLOCK* не меняет поведения файловых дескрипторов. В своё время это было сложно реализовать, а с тех пор стало неявной частью стандарта POSIX. Время ввода с диска может достигать десятков миллисекунд, что, конечно, намного лучше, чем блокировка на

сокете. Тем не менее, именно время ввода с диска стало следующим узким местом в работе высоконагруженных HTTP-серверов.

2.3 Избежание блокирования на дисковом вводе

За последние 10-15 лет было опробовано и успешно применяется несколько механизмов, позволяющих избежать блокирования процесса на дисковом вводе. Рассмотрим их по отдельности.

2.3.1 Выполнение чтения с диска в тредах

Заметим, что в модели «один процесс-одно соединение» блокирование системных вызовов вовсе не является проблемой. Но возвращение к этой модели невозможно из-за огромных издержек на создание процессов. POSIX threads предоставляют нам возможность в рамках одного процесса породить множество контекстов исполнения (тредов), причём порождение тредов существенно менее ресурсоёмко, чем порождение процессов [6]. Соответственно, мы можем передать задачу дискового ввода на выполнение в отдельный тред. В наиболее примитивной модели мы можем создавать по одному треду на каждое соединение. Однако в реальных задачах большинство соединений неактивны, поэтому для осуществления операций с диском достаточно небольшого пула тредов, на несколько порядков меньшего, чем число одновременно обслуживаемых соединений. На момент данной публикации автору неизвестны приложения, реализующие такую модель в чистом виде. Сервер Apache 2 использует треды, однако в нём нет пула тредов, предназначенного именно для работы с вводом/выводом. Вероятно, такой режим работы будет во второй версии сервера nginx.

2.3.2 POSIX Asynchronous I/O

Стандарт POSIX специфицирует ряд функций для осуществления дискового ввода/вывода асинхронно: *aio_read(2)*, *aio_write(2)* и др. [6]. Что если отказаться от *sendfile(2)*, вернуться к циклу из листинга 1.1, но вместо *read(2)*, использовать *aio_read(2)*? Конечно, в чистом виде повторить цикл не удастся, но если сообщение о завершении I/O получать через тот же диспетчер событий, что и сообщения о готовности сокетов, то подобная конструкция отлично вписывается в наш конечный автомат и мы получаем сервер не блокирующийся ни на дисках, ни на сокетах. Такой режим работы есть в nginx.

Необходимо отметить недостатки этого решения. Во-первых, мы отказались от *sendfile(2)*, а значит теперь весь объём пересылаемых данных прогоняется через память приложения. Во-вторых, согласно стандарту POSIX, asynchronous I/O не является обязательным к реализации набором функций. Эти функции отсутствуют во многих операционных системах, или присутствуют с некоторыми ограничениями. Например, в Linux функции AIO реализованы через подсистему Direct I/O, которая работает с дисками напрямую, не пользуясь кэшем виртуальной памяти, что конечно негативно скажется на производительности в большинстве случаев. В-третьих, стоит обратить внимание на оговорку о получении сообщения о завершении асинхронного ввода/вывода. Это тоже реализовано не везде. В полном объёме необходимые интерфейсы реализованы в FreeBSD, однако необходимость копирования данных остаётся узким местом.

2.3.3 *sendfile(2)* с запретом чтения с диска

В 2004 году было предложено небольшое расширение интерфейса *sendfile(2)* в FreeBSD, которое однако привело к выдающимся результатам [12]. Новый флаг *SF_NODISKIO* запрещал системному вызову блокироваться на чтении с диска, то есть *sendfile(2)* может отправить данные в сокет только в том случае, если они в наличии в кэше виртуальной памяти ядра операционной системы. Невозможность слать файл в связи с необходимостью чтения с диска теперь *sendfile(2)* сигнализирует кодом ошибки *EBUSY*. Что же делать приложению, если оно получило такую ошибку? Приложение может инициировать чтение файла с диска в отдельном треде, или с помощью *aio_read(2)*, таким образом подгрузив содержимое файла в кэш операционной системы. Заметим, что кэширование в ядре осуществляется постранично, то есть по 4 килобайта, и если приложение запросит чтение всего 1 байта, закэширована будет полная страница. Первым HTTP-сервером, воспользовавшися новым интерфейсом стал экспериментальный сервер Flash [13], однако вскоре список пользователей *SF_NODISKIO* пополнился Varnish и nginx.

К недостаткам данного метода следует отнести то, что количество системных вызовов необходимых для передачи незакэшированного файла увеличилось приблизительно в два раза. При этом, в процессе передачи процессор переключается между четырьмя контекстами: непосредственно приложение, а также три треда в ядре: тред драйвера сетевой карты обрабатывающий TCP подтверждения от клиента, тред драйвера контроллера диска сообщающий о готовности ввода, и наконец AIO демон (или I/O-тред приложения вместо него). Следует также отметить, что страницы попавшие в кэш виртуальной памяти вследствие *read(2)* или *aio_read(2)* могут быть быстро удалены из кэша в пользу каких-то других данных, то

есть нет гарантий того, что страница пробудет в кэше достаточное время для того, чтобы приложение вызвало *sendfile(2)* повторно. То есть, не исключена ситуация, что повторый вызов *sendfile(2)* опять вернёт *EBUSY*. Тем не менее, несмотря на перечисленные недостатки, именно с помощью этого приёма удавалось добиться максимальной производительности до сегодняшнего дня.

2.3.4 *aio_mlock(2)* перед *sendfile(2)*

Перед тем как приступить к данной работе, автором был опробован ещё один вариант избежания блокирования приложения на диске, сходный с решением из главы 2.3.3, однако исключающий состояние гонки между приложением и процедурой очистки кэша виртуальной памяти от закэшированных страниц. Идея заключается в том, что операционная система предоставляет новый системный вызов *aio_mlock(2)*, то есть асинхронно выполняемый вызов *mlock(2)*. А приложение, перед тем как вызывать *sendfile(2)*, предварительно отображает в свою память отсылаемый регион файла с помощью *mmap(2)* и подкачивает его с помощью *aio_mlock(2)*. После такой подготовки *sendfile(2)* отработает без блокировки, а приложение потом может отдать данные в кэш виртуальной память с помощью *munmap(2)*. Отметим, что отображение файла в память вовсе не означает копирования данных из ядра в память приложения. Если приложение не осуществляет обращений к адресам по отображённому региону, то копирования не происходит.

Новый системный вызов был реализован автором в ревизиях r251523 и r251526 основной ветки FreeBSD [14]. Для HTTP-сервера nginx был разработан патч, задействующий *aio_mlock(2)* в связке с *sendfile(2)*. Сразу отметим необходимость модифицировать приложения под новый API как недостаток идеи.

Однако, результаты оказались неудовлетворительными и идею с *aio_mlock(2)* пришлось признать неудачной. Основная проблема данного варианта заключается в неэффективности системных вызовов *mmap(2)* и *munmap(2)*. Подсистема виртуальной памяти FreeBSD совершенно неоптимизирована под создание и удаление сотен тысяч отображений в памяти одного процесса в секунду. Надо заметить, что это не особенность FreeBSD, а вообще общее свойство современных операционных систем. Такое поведение процесса совершенно нестандартно и ни одна операционная не была задумана так, чтобы быть эффективной под нагрузкой такого рода.

2.3.5 *aio_sendfile(2)*

Также автор рассматривал возможность помещения вызова *sendfile(2)* в подсистему POSIX AIO, подобно тому как туда помещены системные вызовы *read(2)*

и *write(2)* для реализации *aio_read(2)* и *aio_write(2)* соответственно. Однако, от этой идеи пришлось отказаться ещё на этапе реализации. Во-первых, асинхронная работа с сокетами намного сложнее, чем асинхронная работа с дисками, и поэтому в коде AIO демона эти две сущности разделены. А *aio_sendfile(2)* пришлось бы работать одновременно и с сокетом и с диском асинхронно, что совместить очень нетривиально. Во-вторых, профилирование сервера работающего в режиме описанном в главе 2.3.3 показывает, что очереди AIO демона являются узким местом, и мы пожалуй хотим от них избавиться, а вовсе не заменить вызовы к *read(2)* на вызовы к *sendfile(2)* в очередях. В-третьих, опыт из главы 2.3.4 напоминает о том, что новое API потребует адаптации приложений, а этого хотелось бы избежать. Поэтому был сделан вывод, что необходимо делать *sendfile(2)* “асинхронным” оставаясь в рамках того API системного вызова, что уже определён.

Глава 3

Текущая реализация *sendfile()* в FreeBSD

3.1 *read(2)* файла и *write(2)* в сокет

Перед тем как приступить к новой реализации, стоит детально разобраться как работает действующая. Вероятно при первоначальной реализации [8], была разобрана работа *read(2)* над файловым дескриптором и работа *write(2)* над сокетным дескриптором и по результатам был написан ядерный аналог цикла из 1.1.

Системный вызов *read(2)* над файловым дескриптором разворачивается приблизительно в следующий стек ядерных функций:

sys_read() -> *vn_read()* -> *VOP_READ()* -> *ffs_read()* -> GEOM -> драйвер

Рис. 3.1: Ядерный стек системного вызова *read(2)*

Ядро находит vnode, соответствующий данному дескриптору, и вызывает VFS метод *VOP_READ()*, который транслируется в специфичный для файловой системы метод read, в данном случае это FFS. Код файловой системы находит или выделяет страницы памяти, отражающие данный регион читаемого файла. Если страницы невалидны, то их необходимо заполнить. Посылается запрос на чтение в уровень GEOM, реализующий абстракцию над блочными устройствами, который в свою очередь передаёт его в драйвер. По окончании чтения, код файловой системы осуществляет копирование данных из теперь уже валидных страниц в буфер в памяти приложения. После копирования страницы помечаются как незанятые и передаются в кэш виртуальной памяти, где они могут быть закэшированы или освобождены и использованы для других данных.

Системный вызов *write(2)* над дескриптором, отображающим TCP-соединение, разворачивается приблизительно в следующий стек в ядре:

sys_write() -> *sosend()* -> *tcp_usr_send()* -> *tcp_output()* -> *ip_output()* -> *ether_output()* -> код драйвера.

Рис. 3.2: Ядерный стек системного вызова *write(2)*

Находится сокет, соответствующий данному дескриптору, и вызывается *sosend()*. Здесь происходит копирование данных из буфера приложения в цепочку специальных структур сетевого стека, называемых mbuf-ы. Каждый mbuf представляет собой описание буфера и прикреплённый к нему mbuf cluster, где собственно хранятся данные. Цепочка mbufs отсылается в протокольный метод send данного сокета, в данном случае TCP. Далее цепочка mbuf-ов следует по всем уровням сетевого стека до драйвера.

3.2 Цикл *sendfile(2)* в ядре

По результатам разбора работы системных вызовов *read(2)* и *write(2)* очевидно, что наиболее примитивный ядерный цикл *sendfile(2)* будет представлять собой поочерёдный вызов *VOP_READ()* и *sosend()*, где первая будет читать данные в какой-то временный буфер в ядерной памяти, а вторая будет нарезать его и копировать в mbuf-ы. Конечно хочется избежать копирования даже внутри ядра. Поэтому можно сделать так, что *VOP_READ()* никуда не копировал данные из страниц, а mbuf-ы в нашей цепочке будут ссылаться непосредственно на страницы памяти, вместо кластеров. Это потребует запретить системе виртуальной памяти освобождать те страницы, на которые ссылаются mbuf-ы, то есть делать эти страницы *wired*. Для того чтобы избежать состояния гонки между нашим кодом и системой виртуальной памяти, мы должны зарезервировать страницу и объявить её wired ещё до вызова *VOP_READ()*. Полезный побочный эффект от предварительной аллокации заключается в том, что мы можем найти уже валидную страницу в кэше виртуальной памяти и тогда вызов *VOP_READ()* в этом шаге цикла не потребуется вовсе. Следует отметить, что это событие не такое уж и редкое, так как многие файловые системы реализуют алгоритмы read ahead и мы можем обнаружить готовые страницы, прочитанные на предыдущих итерациях цикла. Наконец, так как мы самостоятельно подготавливаем mbuf-ы, то мы можем пропустить *sosend()* и сразу отправить их в протокольный метод send.

```
size_t rem;

for (rem = nbytes; rem > 0;
    rem -= PAGE_SIZE, offset += PAGE_SIZE) {
```

```
        vm_page_t p;
        struct mbuf *m;

        p = vm_page_alloc(vnode, offset, VM_ALLOC_WIRED);
        if (!vm_page_valid(p))
                VOP_READ(vnode, offset, PAGE_SIZE);
        m = m_get();
        m->m_data = p;
        so->so_proto->pru_send(m);
}
```

Листинг 3.1: Элементарный цикл реализующий *sendfile(2)*

3.3 Оптимизация: вложенный цикл

Вышеописанная реализация в FreeBSD успешно прослужила до 2006 года, после чего подверглась оптимизации [15]. Проблема первой реализации заключалось в том, что протокольный код вызывался на каждую страницу, то есть 4 Кб данных. Во-первых, это неоптимально с точки зрения процессорных ресурсов. Во-вторых, фрагментация передачи данных в TCP негативно влияет на работу протокола: окно TCP соединения не растёт достаточно быстро, а большое окно помогает увеличить скорость передачи и справляться с потерями сегментов, не сбавляя скорости передачи. В-третьих, TCP получая всякий раз данные по 4 Кб в случае MTU величиной порядка 1500 байт, будет генерировать каждый третий сегмент сильно меньше максимального размера. Наконец, новые сетевые карты с поддержкой TCP segmentation offload (TSO) требуют от операционной системы отсылки данных максимально большими порциями. Оптимизация заключается в том, чтобы накапливать цепочку mbuf-ов до тех пор пока есть место в буфере сокета, и затем отсылать их за один раз.

```
size_t rem;

for (rem = nbytes; rem > 0;) {
        struct mbuf *m, *mtail;

        m = mtail = NULL;
        while (sbspace(so->so_snd) > 0) {
                vm_page_t p;
```

```
                struct mbuf *m0;

                p = vm_page_alloc(vnode, offset, VM_ALLOC_WIRED);
                if (!vm_page_valid(p))
                        VOP_READ(vnode, offset, PAGE_SIZE);
                m0 = m_get();
                m0->m_data = p;

                if (mtail != NULL)
                        mtail->m_next = m0;
                else
                        m = m0;
                mtail = m0;

                rem -= PAGE_SIZE;
                offset += PAGE_SIZE;
        }
        so->so_proto->pru_send(m);
}
```

Листинг 3.2: Оптимизированный *sendfile(2)* с двойным циклом

С 2006 года код подвергался лишь незначительным оптимизациям и общая идея не поменялась. Реализацию как в листинге 3.2 можно обнаружить в последних вышедших версиях FreeBSD.

Глава 4

Новая реализация *sendfile(2)*

4.1 Асинхронный VOP-метод

Для реализации неблокирующегося *sendfile(2)* в первую очередь нам необходим метод VFS, который не будет "засыпать" в ожидании завершения чтения. Сразу обозначим, что в рамках данной работы мы планируем реализовать такой метод для файловой системы UFS/FFS, оставив за скобками прочие файловые системы, такие как ZFS, NFS.

Блокирующиеся методы работают следующим образом: формируют запрос на ввод/вывод к уровню GEOM, посылают этот запрос и вызывают *bwait()*, которая вызывает функцию *sleep(9)*. То есть контекст, пославший запрос, "засыпает". Пробуждает его специальная функция, которая вызывается по завершении операции ввода/вывода, далее именуемая I/O done callback. Какой именно I/O done callback будет вызван, указывается на этапе формирования запроса на ввод/вывод в параметре *b_iodone*. Для блокирующихся операций в качестве I/O done callback указывается функция *bdone()*. Эта функция комплиментарна к *bwait()*, она осуществляет *wakeup(9)* по тому адресу, на котором заснул тред, пославший запрос на ввод/вывод.

Для реализации неблокирующегося метода мы возьмём существующий блокирующийся метод, найдём то место в коде, где он вызывает *bwait()* и переделаем его следующим образом. Вместо вызова *bwait()* мы сохраним весь текущий контекст данного ввода/вывода в какую-то временную структуру и выставим I/O done callback в свою собственную функцию, которая будет работать с сохранённой структурой, после чего отправим запрос на ввод/вывод в GEOM и сразу же вернёмся из метода VOP.

Таким образом, семантика нового *sendfile(2)* будет следующей. Если сокетный дескриптор находится в блокирующемся режиме и если весь запрошенный регион

файла помещается в буфер сокета, то системный вызов инициирует ввод с диска и немедленно вернёт управление в процесс. Если же указано отправить больше данных чем помещается в сокетный буфер, то системный вызов будет многократно осуществлять цикл ввода/вывода, каждый раз запрашивая столько данных, сколько помещается в сокетный буфер, и между итерациями будет "засыпать" на сокетном буфере, ожидая подтверждения о получении данных удалённой стороной. Только на последней итерации цикла системный вызов не будет дожидаться подтверждения. В случае же неблокирующегося ((O_NONBLOCK)) сокетного дескриптора *sendfile(2)* ограничит объём данных читаемых с диска величиной свободного места в сокетном буфере, инициирует чтение этого количества данных и немедленно вернёт управление в процесс.

4.1.1 ***VOP_READ_ASYNC()*?**

Проследим во что разворачивается вызов *VOP_READ()* от самого начала до уровня GEOM, если нижележащая файловая система это UFS/FFS.

VOP_READ() -> *ffs_read()* -> *breadn_flags()* -> *bstrategy()* -> *bufstrategy()* -> *VOP_STRATEGY()* -> *ufs_strategy()* -> *ffs_geom_strategy()* -> GEOM

Рис. 4.1: Стек *VOP_READ()* для UFS/FFS

Сразу бросается в глаза, что покинув уровень VFS, и войдя в реализацию конкретной файловой системы, снова вызывается VFS метод, а именно *VOP_-STRATEGY()*. Это связано с тем, что файловая система UFS/FFS двухуровневая. Верхний уровень UFS реализует каталоги и имена файлов, файловые флаги. А нижележащий уровень FFS реализует только контейнеры данных - inodes. В принципе, под UFS может располагаться не FFS, а иная файловая система. Хотя в современных версиях FreeBSD альтернатив FFS нет, но например, в NetBSD таковые есть, и есть вероятность, что в FreeBSD они снова появятся. Поэтому данный дизайн нельзя нарушать. Отметим этот момент, как усложняющий реализацию *VOP_READ_ASYNC()*.

Теперь найдём в стеке то место, в котором системный вызов уходит в добровольный "сон", для того чтобы дождаться завершения I/O.

VOP_READ() -> *ffs_read()* -> *breadn_flags()* -> *bufwait()* -> *bwait()* -> *_sleep()*
VOP_READ() -> *ffs_read()* -> *cluster_read()* -> *breadn_flags()* -> *bufwait()* -> *bwait()* -> *_sleep()*

Рис. 4.2: Возможные стеки от *VOP_READ()* до *_sleep()*

Во-первых, сразу отметим, что добровольный "сон" происходит между уровнями UFS и FFS. Пока непонятно усложняет это нашу задачу или наоборот. Во-вторых, таких стеков оказывается два, потому что *ffs_read()* может иногда осуществлять чтение не напрямую, а через "библиотеку" VFS cluster, которая осуществляет эвристический анализ обращений к файловой системе для того, чтобы реализовать опережающее чтение данных (далее readahead). Опережающее чтение данных или readahead это неявно осуществляемая подкачка в кэш данных, которые следуют в файле после запрошенных в данном вызове для того, чтобы оптимизировать скорость выполнения последующих вызовов. Отметим, что наличие двух различных стеков в *ffs_read()* однозначно усложняет поставленную задачу.

Стоит обратить внимание на то, что вообще метод *VOP_READ()* есть прямое отображение системного вызова *read(2)* в ядро, и спроектирован он для того, чтобы копировать данные в буфер в памяти приложения. Как уже было отмечено в главе 3.2, реализация *sendfile(2)* использует побочный эффект заполнения страниц от *VOP_READ()*, а вовсе не его непосредственную функцию. Возможно стоит рассмотреть какие-то альтернативы попытке сделать асинхронный вариант *VOP_READ()*?

4.1.2 *sendfile()* работающий поверх пейджера виртуальной памяти

Итак, *sendfile()* поочерёдно обращается к страницам файла, и если они невалидны, то ему требуется, чтобы подсистема виртуальной памяти тем или иным образом сделала страницу валидной, то есть заполнила актуальными данными с диска. Это очень напоминает работу приложения с memory mapped file, то есть регионом памяти возвращённым системным вызовом *mmap(2)*. Фактически, с точки зрения идеологии виртуальной памяти Mach, а виртуальная память в FreeBSD прямой наследник Mach, было бы куда правильнее если бы *sendfile()* обращался напрямую к пейджеру, а не притворялся процессом читающим файл. Единственный очевидный недостаток это потеря эвристики readahead, которая реализована в FFS. Однако, как было показано в главе 2.3.3, современные HTTP-серверы предпочитают не читать данные с диска непосредственно с помощью *sendfile()*, а значит readahead самим *sendfile()* и не инициируется вовсе. Файловый пейджер в FreeBSD не имеет эвристики реализующей readahead, однако предоставляет в своём API некоторые подсказки, которые могут быть использованы для того, чтобы реализовать некоторый примитивный readahead уровнем выше. Когда у пейдже-

ра запрашивается наличие страницы по заданному смещению, то пейджер заодно сообщает сколько страниц до этой страницы и после этой страницы расположены в том же блоке файловой системы, а значит могут быть подняты с диска в рамках одной операции чтения. Мы можем воспользоваться этими подсказками и поднимать страницы с диска не по одной, а максимально большими порциями, а также осуществлять readahead в том случае, если в буфере сокета закончилось место, но приложение требует посылать файл и далее.

Руководствуясь этими соображениями, автор переписал код *sendfile()*, таким образом, что для наполения страниц используется не *VOP_READ()*, а *VOP_-GETPAGES()*. Причём, VOP-метод вызывается не напрямую, а через интерфейс пейджера виртуальной памяти - *vm_pager_getpages()*. С новым кодом можно ознакомиться в ревизии r258815 FreeBSD, в экспериментальной ветви projects/sendfile [16]. Интересным фактом оказалось то, что число операций ввода/вывода у *sendfile()* работающего через интерфейс пейджера оказалось меньше, чем у *sendfile()* в основной ветви FreeBSD. Однако, как было ранее продемонстрировано автором в ревизии r258646, также в экспериментальной ветви projects/sendfile, код *sendfile()* работающего с *VOP_READ()* может быть значительно оптимизирован в сторону уменьшения количества операций ввода/вывода [17]. Стоит отметить, что в данном абзаце под одной операцией ввода/вывода подразумевается вызов из кода *sendfile()* к VFS, а глубже в стеке ввода/вывода такой вызов может порождать несколько запросов к дискам. Поэтому не стоит ориентироваться на эти цифры как на конечный показатель производительности, но с другой стороны и не стоит отметать полезность объединения множества последовательных запросов от более высокого уровня к более низкому в один запрос.

Отметим ещё одно архитектурное преимущество реализации *sendfile()* поверх пейджера. В 2013 году *sendfile(2)* в FreeBSD научился работать не только над файловыми дескрипторами, которые внутри ядра отображаются в объект типа vnode, но и над дескрипторами отображающими участок POSIX shared memory, то есть объекта памяти, выделенного с помощью *shm_open(2)* [18]. Это полезное нововведение позволяет посылать данные из приложения в сеть без их предварительного копирования в пространство памяти ядра. Однако, дескрипторы shared memory не поддерживают *read(2)*, поэтому код *sendfile(2)* был переделан так, чтобы различным образом работать с файловыми и shm-дескрипторами. Код же нового *sendfile(2)* поверх пейджера становится инвариантным по отношению к типу дескриптора, так как работает он теперь не с vnode, а с соответствующим ему vm_object и пейджером данного объекта. Это значительно упрощает код и делает его архитектурно чище.

4.1.3 *VOP_GETPAGES_ASYNC()*

Сравним стеки от *VOP_GETPAGES()* до GEOM и от *VOP_GETPAGES()* до точки сна, в случае если пейджер работает с UFS/FFS:

VOP_GETPAGES() -> *vnode_pager_local_getpages()* -> *vnode_pager_local_getpages0()* -> *vnode_pager_generic_getpages()* -> *bstrategy()* -> GEOM

Рис. 4.3: Стек от *VOP_GETPAGES()* до GEOM

VOP_GETPAGES() -> *vnode_pager_local_getpages()* -> *vnode_pager_local_getpages0()* -> *vnode_pager_generic_getpages()* -> *bwait()* -> *_sleep()*

Рис. 4.4: Стек от *VOP_GETPAGES()* до *_sleep()*

Итак, в функции *vnode_pager_generic_getpages()* формируется запрос на чтение, который отправляется в GEOM с помощью *bstrategy()*, и тут же тред "засыпает" в *bwait()*. Для того, чтобы реализовать *VOP_GETPAGES_ASYNC()*, нам необходимо разделить функцию *vnode_pager_generic_getpages()* на две части. В первую часть попадёт весь код, который сейчас до *bstrategy()*, а во вторую, тот код, который выполняется после *bwait()*, то есть после завершения ввода/вывода. Вторую функцию мы назовём *vnode_pager_generic_getpages_done()*. В том случае когда страницы запрашиваются синхронно, через *VOP_GETPAGES()*, как и прежде будет вызываться *bwait()*, а затем *vnode_pager_generic_getpages_done()*. Для асинхронного API мы будем выставлять *b_iodone* в свою собственную функцию *vnode_pager_generic_getpages_done_async()*, которую тред драйвера будет вызывать по завершении ввода/вывода. Эта функция будет вызывать *vnode_pager_generic_getpages_done()*, а затем вызывать I/O done callback, который был задан как аргумент *VOP_GETPAGES_ASYNC()*, для того, чтобы известить пользователя нового API (в нашем случае код *sendfile()*) о завершении ввода.

Именно так автором и был реализован *VOP_GETPAGES_ASYNC()* для пейджера по умолчанию, который используется UFS/FFS, в ревизии r274914 основной ветки FreeBSD [19]. Для прочих файловых систем *VOP_GETPAGES_ASYNC()* реализован через *VOP_GETPAGES()*, то есть ввод совершается по-прежнему блокируясь, после чего вызывается I/O done callback.

4.2 “Неготовые” данные в сокетных буферах

Задействовав новый инструмент *VOP_GETPAGES_ASYNC()* в коде *sendfile()*, мы сможем избежать блокировки на дисковом вводе. Для этого нам потребуется и код *sendfile()* разделить на две части: часть до дискового ввода/вывода, и часть после него. Вторая часть будет передаваться как аргумент I/O done callback к *VOP_GETPAGES_ASYNC()*, и следовательно выполняться в контексте дискового драйвера. Такая наивная реализация предполагает перемещение вызова *so->so_proto->pru_send()* в I/O done callback. Но тут мы сталкиваемся с очень серьёзной проблемой. Что если операции ввода/вывода будут завершаться не в том порядке, в каком мы их отправляли? Что если файл уже частично отображён в память и валидные страницы следуют вперемежку с невалидными? Наконец, что если приложение сделает *sendfile(2)* в сокет и сразу же *write(2)*? Ответ на все эти вопросы один и тот же: данные будут попадать в сокет не в том порядке, в каком следует. Передаваемый файл будет испорчен: данные в нём могут быть перемешаны.

Очевидно, что если мы будем отправлять данные в сокет не из I/O done callback, а сразу после *VOP_GETPAGES_ASYNC()*, то ситуация будет ещё печальнее - в сеть уйдут пакеты со случайным содержимым.

То есть, нам нужно сразу после *VOP_GETPAGES_ASYNC()* отправлять данные в сокет, где они надёжно займут положенное им место в буфере отправки, но не позволять им быть отправленными по сети, до тех пор, пока ввод/вывод не завершится. Это потребует архитектурных изменений непосредственно к уровню сокетных буферов, а также к протоколам, с которыми работает *sendfile(2)*. В буферах должна появиться концепция “неготовых” данных, то есть таких данных, которые занимают своё место в цепочке, учитываются в счётчиках занятой буферной памяти, но при этом протоколам они доступны лишь частично. Протокол не может отправить такие данные в сеть, но при этом знает, что они есть в буфере. Когда дисковый ввод завершается, то мы изменяем сокетный буфер, помечаем данные как “готовые”, увеличиваем счётчик байт доступных к отправке и мы должны каким-то образом разбудить протокол, известить его о том, что хотя новых данных в сокетном буфере и не появилось, но тем не менее, ему есть что отправить в сеть.

4.2.1 Сокетный буфер

В сетевом стеке BSD сокетный буфер представляет собой цепочку mbuf-ов. Это типичный FIFO, в буфере есть указатель на первый mbuf в цепочке, и указатель

на последний. Новые данные записываются в буфер после последнего mbuf, а протокол вынимает данные из буфера начиная с первого mbuf-а.

```
struct sockbuf {
        struct mbuf *sb_mb;      /* first mbuf in chain */
        struct mbuf *sb_mbtail;  /* last mbuf in chain */
        u_int sb_cc;             /* chars in buffer */
};
```

Листинг 4.1: Интересующие нас поля в *struct sockbuf*

Для того, чтобы не пересчитывать длину цепочки всякий раз, сумма длин всех mbuf-ов хранится в поле *sb_cc* (char count). В поле *sb_cc* смотрят как подсистемы, пишущие в буфер, для того, чтобы понять осталось ли в нём ещё свободное место для записи, так и подсистемы, читающие из буфера, чтобы выяснить есть ли данные, которые можно отправить.

4.2.2 Сокетный буфер с "неготовыми" данными

Предлагаемая концепция "неготовых" данных подразумевает, что теперь на вопросы "сколько записано в сокетный буфер?" и "сколько можно прочесть из буфера?" можно получить разные ответы. А значит одного поля *sb_cc* уже недостаточно для агрегации информации о длине данных в цепочке mbuf-ов. Введём два новых поля *sb_acc* (available char count) и *sb_ccc* (committed char count) вместо одного. Первое будет обозначать сколько байт можно прочитать из буфера, а второе сколько байт уже записано в буфер. Очевидно, что $sb_acc \leq sb_ccc$.

Также нам потребуется ввести новый флаг для mbuf, который будет помечать mbuf содержающий неготовые данные. И отдельный флаг для готовых mbuf, но которые находятся в FIFO очереди позади как минимум одного неготового, а значит они заблокированы. Назовём эти флаги *M_NOTREADY* и *M_BLOCKED*. Если $sb_acc \neq sb_ccc$, то очевидно, что в буфере есть неготовые данные, и обратное также строго верно. Чтобы избегать всякий раз поиска первого неготового mbuf, введём ещё один указатель в *struct sockbuf*.

```
struct sockbuf {
        struct mbuf *sb_mb;      /* first mbuf in chain */
        struct mbuf *sb_mbtail;  /* last mbuf in chain */
        struct mbuf *sb_fnrdy;   /* first not ready mbuf */
        u_int sb_acc;            /* available char count */
        u_int sb_ccc;            /* committed char count */
```

```
};
```

Листинг 4.2: *struct sockbuf* модифицированный для работы с неготовыми данными

В код, работающий с сокетными буферами нам потребуется внести следующие изменения:

- При добавлении mbuf-ов в очередь, правильно увеличивать sb_acc и sb_ccc, в зависимости от флагов этих mbuf, а также при необходимости менять sb_fnrdy.
- При удалении mbuf-ов, правильно уменьшать sb_acc и sb_ccc, изменять sb_fnrdy и избегать освобождения mbuf-ов с флагом *M_NOTREADY*, так как на них всё ещё ссылаются операции ввода/вывода.
- Реализовать функцию *sbready()*, которая будет убирать флаг *M_NOTREADY* с заданного числа mbuf-ов в цепочке. Если эти mbuf-ы блокировали какие-то готовые данные, то с них необходимо убрать флаг *M_BLOCKED*. Если готовым стал mbuf, на который указывает sb_fnrdy, то следует увеличить sb_acc и обновить значение sb_fnrdy.

Данные изменения реализованы автором в ревизии r275326 основной ветки FreeBSD [20].

4.2.3 Работа сетевых протоколов с “неготовыми” данными

Согласно спецификации системный вызов *sendfile(2)* работает над любыми сокетами типа *SOCK_STREAM*. То есть это TCP-сокеты, локальные сокеты UNIX и потоковые сокеты SCTP. Так как на сегодняшний день неизвестно практического использования *sendfile(2)* над сокетами SCTP, и даже отсутствуют регрессионные тесты данного функционала, то автор решил вынести этот вопрос за рамки данной работы. Таким образом нам необходимо научить работе с неготовыми данными в сокетных буферах TCP сокеты и локальные сокеты. API протоколов потребуется изменить следующим образом:

- Метод *pru_send* должен обзавестить новым флагом *PRUS_NOTREADY*, указывающим на то, что в данной транзакции поступают неготовые данные.
- Необходим новый метод *pru_ready*, который будет будить протокол в том случае, когда новых данных в сокетном буфере не появилось, но sb_fnrdy сместился вперёд по цепочке.

Расширение протокольного API представлено автором в ревизии r275329 [21].

Модификация TCP для работы с неготовыми данными оказалась достаточно простой. Всё, что требуется, это во-первых, не вызывать *tcp_output()* при поступлении неготовых данных через *pru_send*, и во-вторых, вызывать *tcp_output()* когда *pru_ready* активирует ненулевое количество данных в буфере отправки. С изменениями к протоколу TCP можно ознакомиться в ревизии r275333 [22].

С *AF_LOCAL* дела обстоят несколько сложнее. Дело в том, что у локальных сокетов нет буфера отправки как такового. Действительно, зачем он нужен, если сокеты локальны и соединены один к одному? При записи в локальный сокет данные сразу попадают в буфер получения противоположного сокета. То есть, *pru_send* должен отрабатывать как обычно. Если записанные данные неготовы, то sb_acc буфера получения противоположного сокета не увеличится и процессы следящие за ним не будут оповещены. А *pru_ready* также должен работать напрямую с противоположным сокетом. С изменениями к сокетам семейства *AF_LOCAL* можно ознакомиться в ревизии r275332 [23].

4.3 Новый *sendfile(2)*

Итак, владея новыми API в сетевом стеке, VFS и виртуальной памяти, мы можем приступить к решению нашей основной задачи. Основой алгоритма станет двойной цикл из 3.2. Напомним, что мы уже проделали некоторые важные изменения над этим кодом в рамках подготовки. Модифицировали цикл таким образом, что запрос к VFS вынесен во внешний цикл [17]. Стали обращаться к пейджеру для подкачки страниц с диска [16].

4.3.1 *sendfile_swapin()*

Весь код связанный с подкачкой страниц вынесен в отдельную функцию, названную *sendfile_swapin()*. Функция работает с объектом и вектором страниц расположенных по заданному смещению в объекте, то есть она инвариантна по отношению к сетевому стеку и типу объекта, будь то файл или сегмент shared memory. Фактически эта функция может быть перенесена в подсистему виртуальной памяти, если у неё найдутся ещё пользователи кроме *sendfile()*. Функция получает на вход вектор из указателей на страницы, которые она должна инициализировать, а затем и заполнить эти страницы данными. Сначала функция заполняет вектор, каждая страница берётся у виртуальной памяти как wired и busy:

```
for (int i = 0; i < npages; i++)
```

```
        pa[i] = vm_page_grab(obj, page_offset(i),
            VM_ALLOC_WIRED);
```

Листинг 4.3: *sendfile_swapin()*: инициализация вектора страниц

Затем вектор страниц обрабатывается следующим алгоритмом:

- Если страница уже валидна (находится в кэше виртуальной памяти), снимаем бит busy и переходим к следующей.
- Если страница невалидна, то в отдельном цикле ищем следующую валидную страницу. Таким образом мы получаем последовательность идущих подряд невалидных страниц.
- Пейджер опрашивается о наличии невалидных страниц. Для файлового пейджера отсутствие страницы возможно только в случае если файл с “дырами” (sparse file). Отсутствующая страница заполняется нулями и перестаёт быть busy. При наличии страницы пейджер также даёт нам подсказку о том, сколько страниц после этой страницы мы можем поднять с диска за одну операцию ввода.
- Вычисляется минимум между необходимым количеством страниц и подсказкой пейджера, и это количество страниц запрашивается у пейджера.

```
for (int i = 0; i < npages;) {
        int j, after, count;

        if (vm_page_is_valid(pa[i]) {
                vm_page_xunbusy(pa[i]);
                i++;
                continue;
        }

        for (j = i + 1; j < npages; j++)
                if (vm_page_is_valid(pa[j])
                        break;

        while (!vm_pager_has_page(obj, pa[i],i &after) &&
            i < j) {
                pmap_zero_page(pa[i]);
                pa[i]->valid = 1;
```

```
                    vm_page_xunbusy(pa[i]);
                    i++;
            }
            if (i == j)
                    continue;

            count = min(after + 1, npages - i);
            refcount_acquire(&sfio->nios);
            vm_pager_get_pages_async(obj, pa + i, count,
                sf_iodone);
            vm_page_xunbusy(pa[i]);
            i += count;
    }
```

Листинг 4.4: *sendfile_swapin()*: подкачка страниц из пейджера

4.3.2 Основной цикл

Для того, чтобы сохранить состояние между контекстом системного вызова и I/O done, мы будем резервировать память под специальную структуру *struct sf_io*. В структуре мы будем сохранять указатель на сокет, указатель на первый отправленный mbuf, содержащий неготовые данные, счётчик асинхронных запросов к пейджеру в рамках данного системного вызова, а также вектор страниц, с которым работает *sendfile_swapin()*. Так как длина вектора переменна, то и размер структуры также переменный.

```
struct sf_io {
        u_int           nios;
        int             npages;
        struct file     *sock_fp;
        struct mbuf     *m;
        vm_page_t       pa[];
};
```

Листинг 4.5: Структура *struct sf_io*

В каждой итерации внешнего цикла мы вычисляем количество страниц необходимое для заполнения буфера отправки, резервируем память под sf_io соответствующего размера и вызываем *sendfile_swapin()*, которая возвращает число запущенных им асинхронных чтений. Если в процессе подкачки не повстречалось

ни одной невалидной страницы, что вполне вероятно если файл закеширован в памяти, то тогда функция вернёт нуль. Цепочка mbuf-ов формируется также как и ранее, за одним дополнением. Если *sendfile_swapin()* сообщил, что были запущены асинхронные задачи ввода, то мы помечаем всю цепочку как *M_NOTREADY*. Можно было бы отмечать как *M_NOTREADY* только те mbuf-ы, страницам которых потребовалась операция чтения, в таком случае протокол смог бы сразу отослать все страницы до первой неготовой в сеть. Однако, такая реализация была бы сложнее, и как уже было замечено ранее в главе 3.3, выгоднее отсылать данные по возможности более крупными порциями. В конце раунда внешнего цикла мы вызываем *pru_send*, причём, если известно, что мы посылаем неготовые данные, то выставляем флаг *PRUS_NOTREADY*.

```
for (off = offset; rem > 0; ) {
        struct mbuf *m, *mtail;
        int space, npages;

        space = sbspace(&so->so_snd);
        npages = howmany(space + (off & PAGE_MASK), PAGE_SIZE);
        sfio = malloc(sizeof(*sfio) + npages * sizeof(vm_page_t));
        refcount_init(&sfio->nios, 1);
        nios = sendfile_swapin(obj, sfio, off, space, npages);
        m = mtail = NULL;
        for (int i = 0; i < npages; i++) {
                struct mbuf *m0;

                m0 = m_get();
                m0->m_data = sfio->pa[i];
                if (nios)
                        m0->m_flags |= M_NOTREADY;
                if (mtail != NULL)
                        mtail->m_next = m0;
                else
                        m = m0;
                mtail = m0;
        }

        off += space;
        rem -= space;
```

```
        if (nios) {
                so->so_proto->pru_send(m, PRUS_NOTREADY);
                sf_iodone(sfio);
        } else
                so->so_proto->pru_send(m);
}
```

Листинг 4.6: Основной цикл нового *sendfile()*

4.3.3 Завершение чтения: *sf_iodone()*

Всякий раз инициируя асинхронный запрос на чтение *sendfile_swapin()* указывает как I/O done callback функцию *sf_iodone()*. По нашей задумке эта функция должна вызывать протокольный метод *pru_ready*. Однако, мы можем его вызвать тогда и только тогда, когда все асинхронные вызовы для данной цепочки mbuf-ов будут завершены, а также когда будет завершён раунд основного цикла, который инициировал все эти чтения. Теоретически такое может быть, что чтение с диска обгонит выполнение цикла. Поэтому перед всяким асинхронным вызовом пейджера мы будем увеличивать на единицу счётчик nios внутри sf_io. Кроме того, сам раунд цикла тоже удерживает одну ссылку на эту sf_io. Контекст системного вызова уберёт эту ссылку тогда, когда завершит формировать цепочку mbuf-ов и будет готов её отправить в протокол. Таким образом, *sf_iodone()* должна начинаться с уменьшения счётчика ссылок и проверки его значения. Если счётчик не достиг нуля, функция возвращается. И только последний вызов *sf_iodone()* для данного sf_io сможет вызывать протокольный *pru_ready*.

```
if (!refcount_release(&sfio->nios))
        return;
so->so_proto->pru_ready(sfio->so, sfio->m);
free(sfio);
```

Листинг 4.7: *sf_iodone()*

На момент публикации данной работы новая реализация *sendfile(2)* ещё не перенесена в основную ветвь разработки FreeBSD. С кодом реализующим идеи, описанные в данной главе, можно ознакомиться в экспериментальной ветке projects/sendfile, файл sys/kern/uipc_syscalls.c, функции *vn_sendfile()*, *sendfile_swapin()* и *sf_iodone()* [24].

4.4 Оптимизация и расширение API нового *sendfile(2)*

Новый код, работающий напрямую с пейджером, позволяет более тонко контролировать работу виртуальной памяти со страницами, которые поднимаются с диска в результате работы *sendfile(2)*. Например, мы можем запретить виртуальной памяти кэшировать эти страницы, или мы можем настраивать величину readahead вручную, что было невозможно при подкачке с помощью *VOP_READ()*. Мы можем отдать контроль над этими расширениями приложению, введя новые флаги для системного вызова.

4.4.1 Запрет кэширования

В большинстве случаев, кэширование виртуальной памятью файлов весьма полезно, однако, можно представить себе такие инсталляции когда издержки на кэширование выше, чем реальный выигрыш от него. Например, современный сервер отдающий по HTTP большую коллекцию видеофайлов имеет дисковый объём порядка 100 терабайт, а объём оперативной памяти порядка 64 гигабайт, то есть объём дисков на три с лишним порядка превышает объём оперативной памяти [25]. При абсолютно случайном доступе к файлам, вероятность попадания в кэш будет составлять порядка 0.1%. Конечно, на практике не бывает абсолютно случайного доступа, какой-то контент популярнее. Практически измеренные результаты показывают попадаемость в кэш на уровне 2-5%. При этом кэш не бесплатен. Через него течёт большой трафик страниц: свежие страницы постоянно добавляются, старые удаляются. Это требует поддержания списков, синхронизации доступа между процессорами и пр. Просто возвращать страницы в пул свободных страниц было бы дешевле.

Тем временем оператор сервера знает какой контент популярен, а какой нет. Если проинформировать HTTP-сервер о популярности файлов, или сделать так, чтобы сервер рассчитывал её динамически, то HTTP-сервер мог бы указывать ядру о том, какие вызовы к *sendfile(2)* должны кэшироваться, а какие нет. При этом мы сможем не только избежать расхода лишних процессорных ресурсов, но и существенно поднять процент попадания в кэш для действительно популярных файлов.

Новый *sendfile(2)* понимает флаг *SF_NOCACHE*, который внутри ядра транслируется в иную функцию освобождения mbuf-ов, которая освобождает страницу, принадлежащую mbuf-у, сразу в пул свободных страниц, мимо кэша. Заметим, что для старой реализации *sendfile(2)* это не сработало бы, так как ранее mbuf не являлся последней ссылкой на страницу.

4.4.2 Приложение указывает readahead

Интерфейс пейджера *vm_pager_has_page()* даёт нам подсказку сколько страниц мы можем подкачать бесплатно вместе с запрошенной. Эта подсказка используется в том случае если системный вызов знает, что это не последний раунд внешнего цикла (см. листинг 4.4). В последнем же раунде мы не знаем стоит ли делать readahead или нет. Неизвестно будут ли ещё системные вызовы на отправку последующих страниц этого же файла. У приложения, в частности HTTP-сервера, такая информация может быть. Опять же ссылаясь на задачу отдачи видео, заметим, что HTTP-сервер может извлечь из видео-файла информацию о его структуре и с ее помощью прогнозировать дальнейший доступ клиента к файлу. Таким образом, было бы полезно позволить приложению указывать дозволенный readahead. Для этого новый *sendfile(2)* рассматривает старшие 16 бит флагов как количество страниц readahead. Макрос *SF_FLAGS()* позволяет скомбинировать значение readahead и обычные флаги.

```
#define SF_FLAGS(rh, flags)      (((rh) << 16) | (flags))
```

Листинг 4.8: Макрос *SF_FLAGS()*

Глава 5

Анализ результатов

5.1 Сравнение существующей и предлагаемой реализаций в прикладной задаче

Для автора критерием успешности данной работы является сравнение новой реализации и старой в реальной работе серверов отдачи потокового видео компании Netflix [26]. Такой сервер представляет собой современную машину архитектуры amd64 с большим количеством HDD дисков или SSD устройств, либо и тех и других. Сервер оснащён несколькими десятигигабитными сетевыми картами и несколькими десятками гигабайт памяти. Более подробное описание одной из используемых платформ приведено в главе 5.2. Шаблон запросов от клиентов далёк от оптимального: средний размер запроса 100 Кб, доступ к файлам достаточно случайный. Параллельно сервер обслуживает порядка 20 - 30 тысяч клиентов.

Критерием качества работы сервера является то, сколько клиентов он способен обслуживать и сколько трафика способен отдавать без деградации качества сервиса.

Новая реализация показала следующие преимущества перед старой:

- Увеличение пиковой производительности сервера до 20%. Количественные результаты сильно варьируются от модели сервера (HDD или SSD, вычислительная мощность CPU) и от профиля клиентского трафика. Но на качественном уровне новая реализация всегда оказывается более производительной, чем старая.

- Отсутствие всплесков занятости процессора, более ровный профиль загрузки при максимальной нагрузке. Старая реализация требовала ограничивать загрузку сервера до 80% его мощности, чтобы избежать деградации сервиса

при спорадических пиках загрузки CPU. Новая реализация позволяет безопасно поднять эту планку до 95% загрузки.

- Возможность тонко контролировать кэширование страниц в виртуальной памяти и readahead. Правильная эксплуатация этих расширений API позволила более эффективно распоряжаться оперативной памятью и получить дополнительный прирост производительности сервера отдачи видео.

Следует отдельно отметить, что ещё на этапе постановки задачи было предположено, что как только удастся развязаться с дисковым вводом, то следующими узкими местами станут подсистема виртуальной памяти и TCP стек. В связи с этим, параллельно с данной работой, отдельно велись и работы по устранению прочих, пока только предсказанных узких мест. И действительно, после внедрения новой реализации обнаружились ранее предсказанные узкие места. На момент публикации данной работы, уже готова предварительная версия улучшений к подсистеме виртуальной памяти. Данные улучшения, будучи приложенными к старой блокирующейся на дисковом чтении версии *sendfile(2)*, практически не дают заметного эффекта, в комбинации же с новым *sendfile(2)* удалось увеличить производительность на 50 - 60%. Так, на сервере видео-отдачи Netflix предыдущего поколения производительность выросла с 15 Гбит/с до 25 Гбит/с, а на серверах текущего поколения с 23 Гбит/с до 35 Гбит/с. На рис. 5.1 и 5.2 представлены результаты теста, в котором сравнивалась работа новой реализации *sendfile(2)* в комбинации с экспериментальными улучшениями к подсистеме виртуальной памяти, против старой реализации также с улучшениями к системе виртуальной памяти.

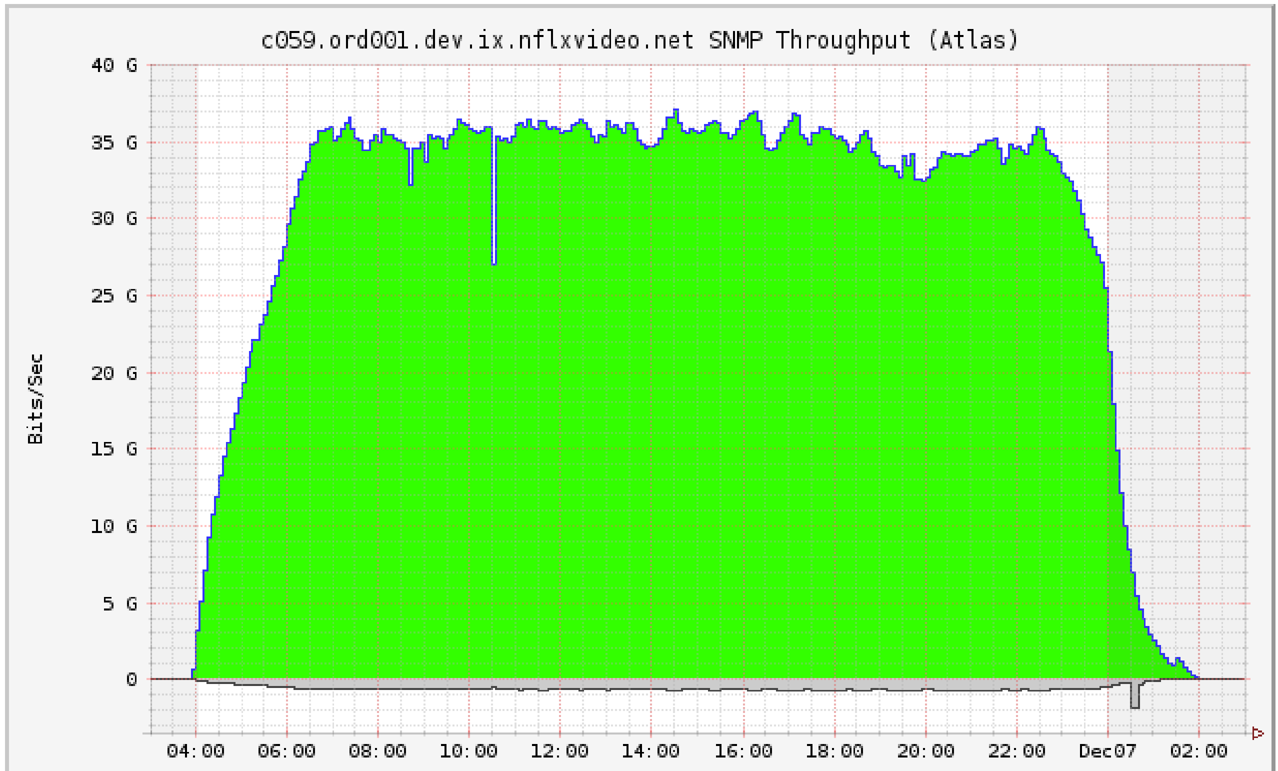

Рис. 5.1: Утилизация сети: новый *sendfile(2)* + экспериментальные улучшения VM

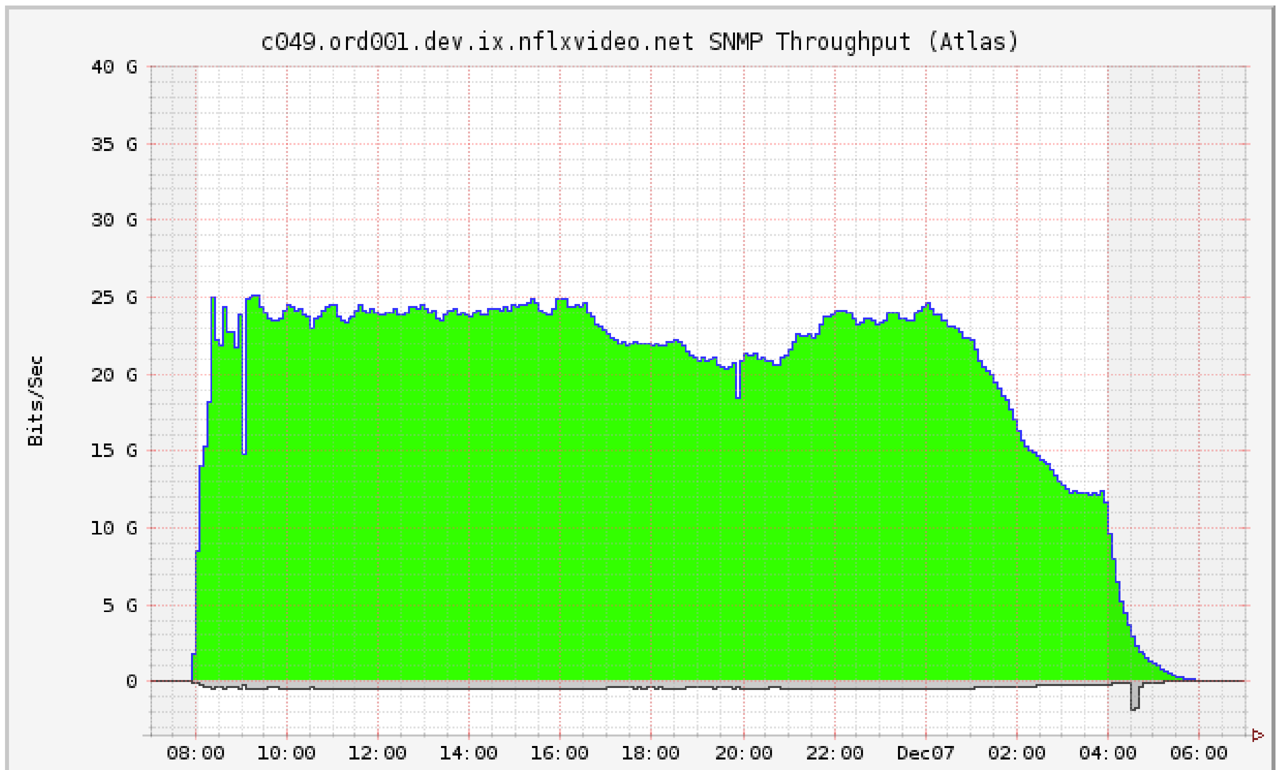

Рис. 5.2: Утилизация сети: только экспериментальные улучшения VM

5.2 Сравнение существующей и предлагаемой реализаций в синтетическом бенчмарке

Для синтетического бенчмарка использовалась следующая платформа:

- Intel® Xeon® CPU E5-2650L, 1.8 ГГц, 8 ядер
- 64 Гб оперативной памяти
- 36 жёстких дисков объёмом 5.4 Тб каждый
- сетевая карта Chelsio T420-CR с двумя портами пропускной способностью 10Гбит/с каждый

На каждом диске было расположено по 4.3 Тб данных в виде файлов случайного размера, варьирующегося от 50 Мб до 1 Гб. Всего было сгенерировано порядка 800 тысяч файлов. На тестируемой машине был запущен HTTP-сервер nginx версии 1.5.12, в файле конфигурации nginx.conf было указан режим работы *sendfile(SF_NODISKIO) + aio_read()*. Поочерёдно загружалось ядро FreeBSD 11 как с новой, так и со старой реализацией *sendfile(2)*. В конфигурацию ядра были внесены следующие изменения:

```
options MAXPHYS (1024*1024)
options DFLTPHYS (1024*1024)
options NSWBUF_MIN 512
options FLOWTABLE
options VFS_AIO
#define PA_LOCK_COUNT 32768
```

Генератором нагрузки служила заведомо более мощная платформа, с процессором Intel® Xeon® E5-2697 v2 и 40 Гбитной сетевой картой. В качестве утилиты для создания нагрузки использовалась wrk [27]. Утилите wrk предоставлялся полный список файлов на тестируемой машине, и она случайным образом выбирала из него файлы для запросов. В параметрах запуска wrk указывалось открывать 100 соединений и 12 тредов (согласно числу ядер на платформе). Перед каждым раундом теста тестируемая машина перезагружалась. Раунд теста длился 10 минут. В течение тестов проводились измерения загрузки CPU и исходящего трафика.

В сравнении рассматривались старая реализация, новая реализация с настройками по умолчанию, и отдельно с принудительным выставлением флага *SF_NOCACHE*, то есть с запретом на кэширование отосланных в сеть страниц в кэше

виртуальной памяти (см. главу 4.4.1). Следует обратить внимание, что соотношение объёма оперативной памяти в тестируемой платформе (64 Гб) к общему размеру файлов на дисках (154 Тб) составляет 0.04%, то есть при случайном доступе смысла кэшировать данные нет. Поэтому следует ожидать от теста с принудительным отключением кэширования меньшей загрузки CPU.

5.2.1 Трафик

В данном тесте как старая, так и новая реализация способны полностью реализовать возможности двух 10 Гбитных сетевых карт. То есть в обоих случаях трафик упирается в 20 Гбит/с (2.4 Гбайт/сек), однако при детальном рассмотрении можно заметить спорадические провалы в производительности, и именно по ним мы можем сравнить старую реализацию с новой. В данном тесте в течение 10 минут старая реализация смогла отдать 1.25 Тбайт данных, а новая 1.28 Тбайт. На графике трафика хорошо видно где были потеряны 30 Гбайт.

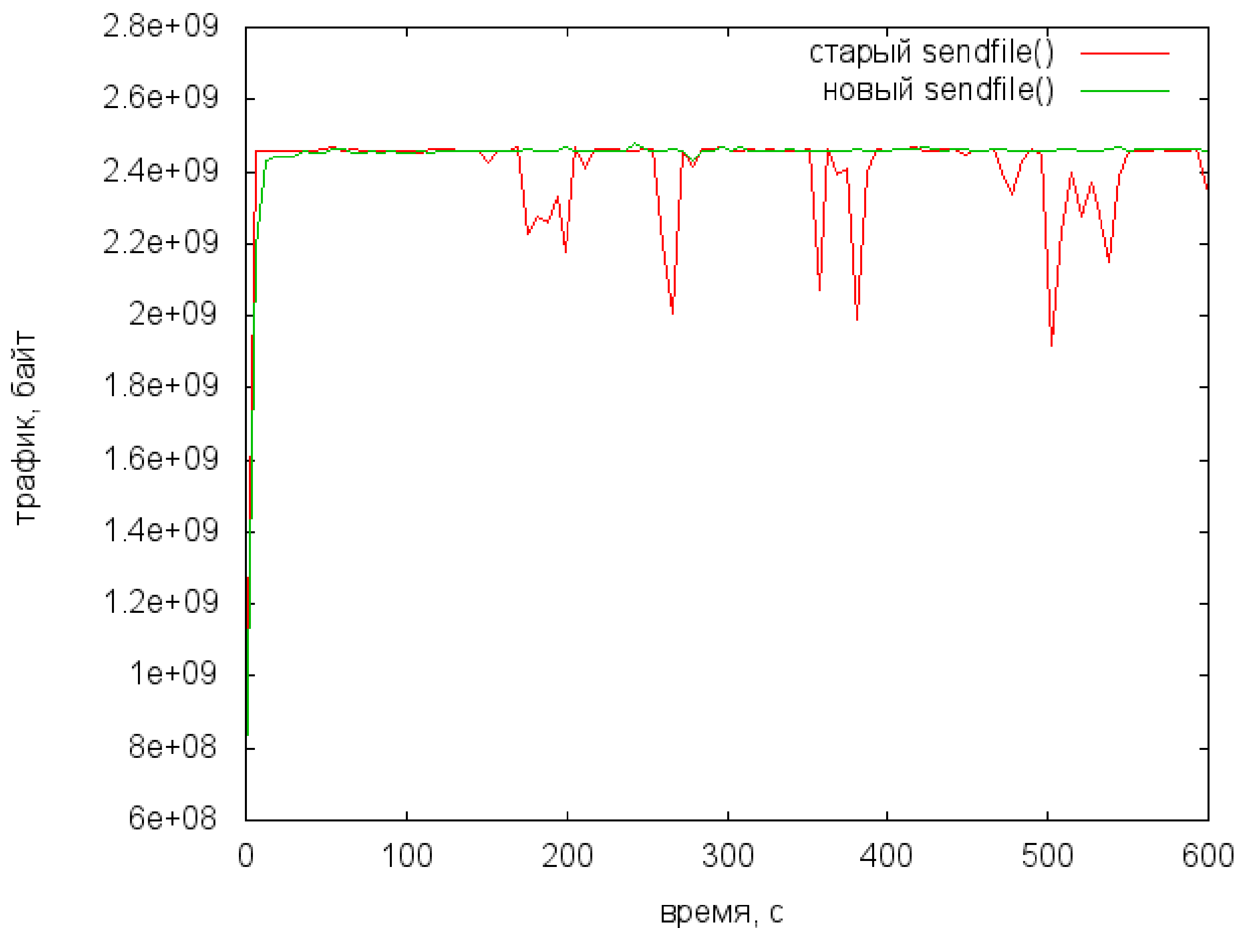

Рис. 5.3: Трафик в 10-ти минутном тесте

5.2.2 Загрузка CPU

В течение 10 минут теста каждую секунду замерялось время простоя процессоров, иными словами свободные вычислительные ресурсы. Чем время простоя больше, тем, очевидно, эффективнее работает код. На рис. 5.4 невооружённым глазом видно, что новая реализация потребляет меньше ресурсов, чем старая, а включение *SF_NOCACHE* освобождает ещё больше ресурсов. После статистической обработки полученных данных можно заявить с достоверностью 95%, что в данном испытании новая реализация потребила на 11.64% меньше процессорных ресурсов. Новая реализация с отключенным кэшированием потребила на 20.99% процессорных ресурсов меньше, чем старая реализация.

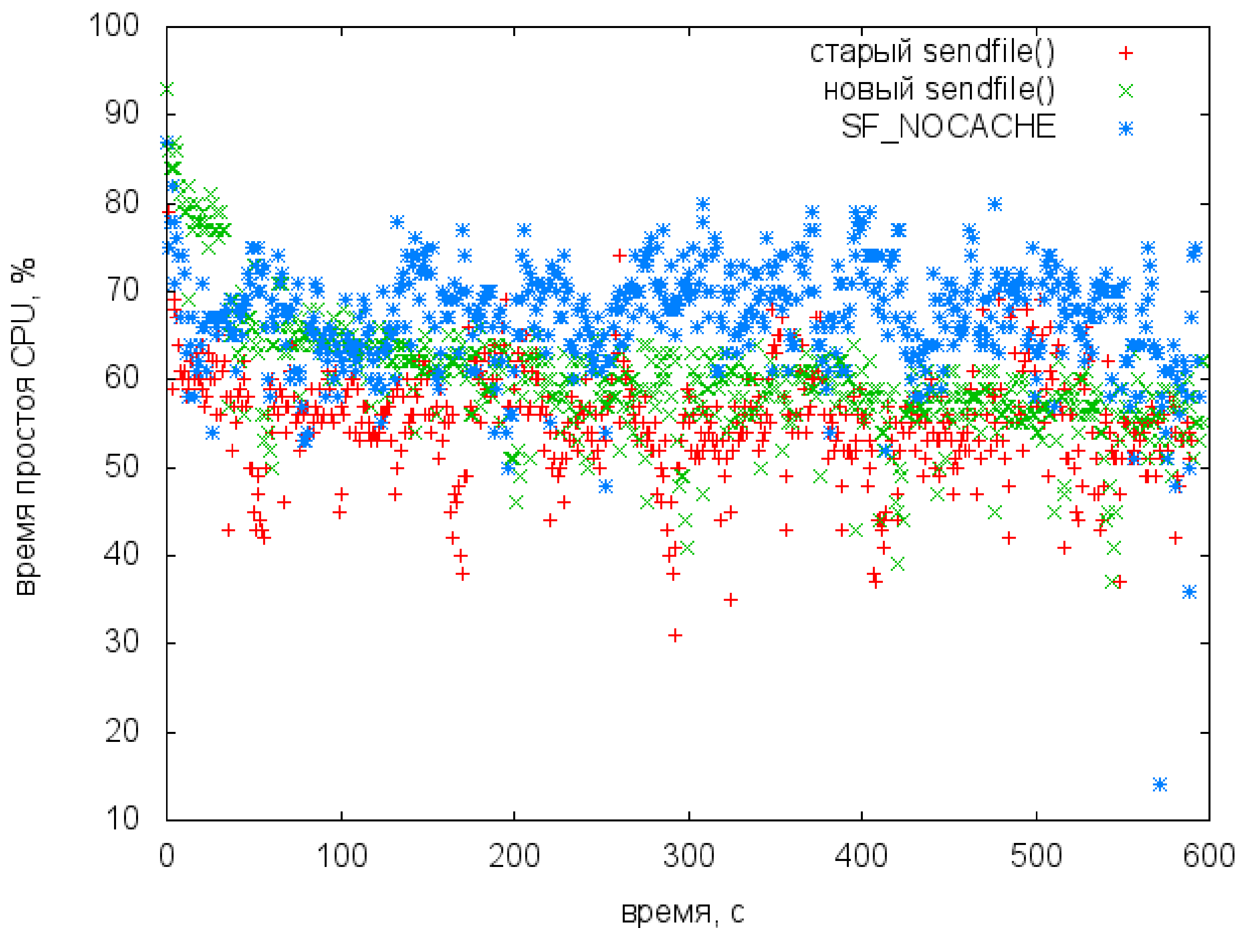

Рис. 5.4: Время простоя процессора в 10-ти минутном тесте

Заключение

В работе решены все изначально поставленные задачи:

- Реализован новый метод *VOP_GETPAGES_ASYNC*, позволяющий подкачивать страницы файла в память асинхронно.
- Реализован новый API пейджеров виртуальной памяти *vm_pager_getpages_-async()*, позволяющий запрашивать у пейджеров страницы асинхронно.
- Осуществлены необходимые изменения в подсистеме сокетных буферов, позволяющие помещать в буфер страницы, подкачка которых ещё не завершилась.
- Опираясь на вышеперечисленные изменения, заново реализован системный вызов *sendfile(2)*. Новая реализация не блокируется на дисковом чтении, и возвращает контроль приложению немедленно.

Таким образом, основная цель работы - избежать блокирования на дисковом чтении в системном вызова *sendfile(2)* успешно решена, причём таким образом, что для поддержки нового функционала не требуется модификации существующих приложений. Попутно автором были найдены способы предоставить приложению больший контроль над страницами виртуальной памяти задействованными в данном системном вызове, что было также реализовано автором в виде расширения API.

Разработанная автором реализация показала увеличение пиковой производительности на 20% на сервере раздачи видео Netflix, причём было продемонстрировано, что потенциал новой реализации этим не ограничивается и составляет порядка 50% - 60%.

Проведено сравнение новой реализации и старой в условиях синтетической нагрузки, что также подтвердило её преимущества и потенциал для дальнейшего развития.

Литература

[1] RFC 765, FILE TRANSFER PROTOCOL
http://tools.ietf.org/html/rfc765

[2] RFC1436, The Internet Gopher Protocol (a distributed document search and retrieval protocol)
http://tools.ietf.org/html/rfc1436

[3] The Original HTTP as defined in 1991
http://www.w3.org/Protocols/HTTP/AsImplemented.html

[4] RFC1945, Hypertext Transfer Protocol – HTTP/1.0
http://tools.ietf.org/html/rfc1945

[5] Sandvine Global Internet Phenomena Report - 1H 2014
https://www.sandvine.com/downloads/general/global-internet-phenomena/2014/1h-2014-global-internet-phenomena-report.pdf

[6] The Open Group Base Specifications Issue 7
IEEE Std 1003.1™, 2013 Edition
http://pubs.opengroup.org/onlinepubs/9699919799/toc.htm

[7] send system call - new system call: sendfile (NcEn316)
http://h21007.www2.hp.com/portal/download/files/unprot/STK/HPUX_STK/impacts/i316.html

[8] FreeBSD, ревизия r40931
David Greenman
http://svnweb.freebsd.org/base?view=revision&revision=40931

[9] The C10K problem
Dan Kegel
http://www.kegel.com/c10k.html

[10] Kqueue: A generic and scalable event notification facility
Jonathan Lemon
http://people.freebsd.org/~jlemon/papers/kqueue.pdf

[11] Comparing and Evaluating epoll, select, and poll Event Mechanisms
Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag
http://www.kernel.org/doc/ols/2004/ols2004v1-pages-215-226.pdf

[12] FreeBSD, ревизия r125586
Mike Silbersack
http://svnweb.freebsd.org/base?view=revision&revision=125586

[13] Flash: An efficient and portable Web server
Vivek S. Pai, Peter Druschel, Willy Zwaenepoel
http://usenix.org/publications/library/proceedings/usenix99/full_papers/pai/pai.pdf

[14] FreeBSD, ревизии r251523 и r251526
Gleb Smirnoff
http://svnweb.freebsd.org/base?view=revision&revision=251523
http://svnweb.freebsd.org/base?view=revision&revision=251526

[15] FreeBSD, ревизия r163913
Andre Oppermann
http://svnweb.freebsd.org/base?view=revision&revision=163913

[16] FreeBSD, ревизия r258815
Gleb Smirnoff
http://svnweb.freebsd.org/base?view=revision&revision=258815

[17] FreeBSD, ревизия r258646
Gleb Smirnoff
http://svnweb.freebsd.org/base?view=revision&revision=258646

[18] FreeBSD, ревизия r255467
Konstantin Belousov
http://svnweb.freebsd.org/base?view=revision&revision=255467

[19] FreeBSD, ревизия r274914
Gleb Smirnoff
http://svnweb.freebsd.org/base?view=revision&revision=274914

[20] FreeBSD, ревизия r275326
Gleb Smirnoff
http://svnweb.freebsd.org/base?view=revision&revision=275326

[21] FreeBSD, ревизия r275329
Gleb Smirnoff
http://svnweb.freebsd.org/base?view=revision&revision=275329

[22] FreeBSD, ревизия r275333
Gleb Smirnoff
http://svnweb.freebsd.org/base?view=revision&revision=275333

[23] FreeBSD, ревизия r275332
Gleb Smirnoff
http://svnweb.freebsd.org/base?view=revision&revision=275332

[24] FreeBSD, ветка projects/sendfile, реализация нового *sendfile(2)*
Gleb Smirnoff
http://svnweb.freebsd.org/base/projects/sendfile/sys/kern/uipc_syscalls.c?annotate=275308#l2287

[25] Netflix CDN and Open Source
Gleb Smirnoff
http://www.slideshare.net/facepalmtarbz2/slides-41343025/10

[26] Netflix Open Connect
https://openconnect.itp.netflix.com

[27] Modern HTTP benchmarking tool
https://github.com/wg/wrk