Лекция 15:
Журналируемые файловые системы
Berkeley Log-Structured File System

Алексей Линёв
Александр Мощук
Кирилл Погорельский

somes slides are adapted from the OS course at the University of Washington

Журналируемые файловые системы

- Стали популярными начиная с ~2002 г.
- Основная идея
  - обновлять метаданные (а возможно, и все данные) по принципу транзакций
    - "все или ничего"
  - если происходит сбой, вы можете потерять небольшую часть своей работы, но содержимое диска сохранит непротиворечивое состояние
    - точнее говоря, вы сможете быстро привести его в непротиворечивое состояние, используя журнал транзакций, и не просматривая все блоки диска в поисках несоответствий
- Существуют детали, различающиеся в конкретных ФС
  - ext3, ReiserFS, XFS, JFS, ntfs

Содержание предыдущих серий…

- Оригинальная файловая система UNIX (Bell Labs)
  - одновременно простая и практичная структура
  - отлично иллюстрирует инженерные компромиссы, часто встречающиеся при проектировании
  - элегантная, но медленная
    - причем производительность падает при увеличении размеров диска
- BSD UNIX Fast File System (FFS)
  - увеличенный размер блоков
  - использование "групп цилиндров"
  - использование информации о параметрах аппаратного обеспечения

Где размещаются данные?

- В рассмотренных нами файловых системах данные располагаются в двух местах:
  - на жестком диске
  - в различных кэшах в оперативной памяти
- Наличие кэшей критически важно для производительности, но одновременно служит возможной причиной повреждения целостности ФС при сбоях
- Основная идея решения:
  - поддерживаем некоторую целостную "оригинальную версию" данных
  - изменения запоминаются в виде последовательной серии обновлений, хранящихся в виде хронологически упорядоченной последовательности в разделе/файле журнала
  - в периоды низкой загрузки применяем обновления в хронологическом порядке к "оригинальной версии" данных, освобождая пространство журнала

Результаты последствий при сбоях ужасны в обоих случаях

- Кэширование необходимо из соображений производительности
- Предположим, сбой произошел в ходе операции создания файла
  - 1. Выделить свободный i-node
  - 2. Создать новую запись в каталоге, указывающую на этот i-node
- В принципе, после сбоя дисковые структуры могут находиться в противоречивом состоянии
  - метаданные были обновлены, а данные – нет
  - данные были обновлены, а метаданные – нет
  - те или другие (или и те и другие) были обновлены частично
- fsck (i-check, d-check) работает *очень* медленно
  - нужно просмотреть каждый блок
  - ситуация ухудшается при увеличении размеров диска

Журнал последовательных изменений (Redo log)

- Протокол (log): файл, допускающий только дописывание, содержит записи о запрошенных операциях. Его изменение производится на основе транзакций:
  - <start t>
    - начинаем транзакцию t
  - <t,x,v>
    - запрошена операция изменения содержимого блока x, его новое значение равно v
      - можно хранить в журнале отличия блоков вместо полных значений
  - <commit t>
    - транзакция t подтверждена – в случае сбоя информация о запрошенной операции не потеряется
- Комментарии
  - операция <commit> относится только к записям журнала – "оригинальная версия" данных не изменяется в этот момент
  - не гарантируется целостность содержимого файла (с точки зрения приложений) – мы описали лишь общие положения подхода

Действия в случае сбоев

- Восстанавливаем журнал последовательных изменений
- Выполняем запрошенные операции, описанные в подтвержденных записях
  - Просматриваем журнал в хронологическом порядке и выполняем все операции, описанные подтвержденными записями
  - Отметим, что операция записи идемпотентна: может быть выполнена произвольное положительное число раз с одинаковым конечным результатом
- Операции из неподтвержденных записей
  - Игнорируем – в результаты теряются операции, запрошенные в последние мгновения перед сбоем

Нет ли тут тупиков?

- Почему они могут появиться?
- Что делать?

Управление пространством журнала

- Поток – сборщик изменений просматривает журнал в хронологическом порядке, изменяя "оригинальные версии" данных согласно запросам в подтвержденных записях
  - идемпотентность в данном случае очень важна, поскольку во время выполнения сборки тоже может произойти сбой
- Как только запрос был обработан, содержащая его запись может быть удалена из журнала

- Журналируемые файловые системы
- Berkeley Log-Structured File System

somes slides are adapted from the OS course at the University of Washington

Влияние на производительность

- Журналирование – это последовательная запись данных большого объема
  - очень эффективно
- А с вашей точки зрения, за раз выполняется меньшее количество операций записи
  - и это заметно влияет на производительность
- Таким образом, журналируемая файловая система действительно может сильно повысить производительность
- А также обеспечить эффективное восстановление после сбоев

Еще раз о кэшировании (относится к FS и FFS)

- Кэш (часто называемый *буферным кэшем*) – это просто часть оперативной памяти
- Это общесистемный объект, разделяемый всеми проектами
- Требует использования алгоритма замещения
  - обычно используется LRU
- Даже при использовании кэша небольшого размера сильно повышает эффекттивность
- Настоящее время – время больших объемов оперативной памяти → больших кэшей → больших процентов попадания в кэш
- Многие файловые системы используют технику "опережающего считывания" ("read-ahead"), еще больше повышая производительность

Кэширование при записи и кэширование при чтении

- Некоторые приложения полагают, что по завершении операции записи данные располагаются на диске (звучит достаточно логично!)
- Файловая система может испытывать серьёзные проблемы с целостностью при возникновении сбоев в периодах между выполнениями операции sync – содержимое i-node'ов и блоков данных может не перенести такого испытания
- Подходы
  - "write-through" – "сквозная запись" буферного кэша (синхронная – и медленная)
  - "write-behind" – "отложенная запись", поддерживается очередь еще не записанных блоков, которые периодически записываются на диск (ненадежная схема, рассчитанная на использование sync)
  - NVRAM (энергонезависимая память) – запись производится в память с аварийным питанием от батарей (дорого), а позднее – на жесткий диск

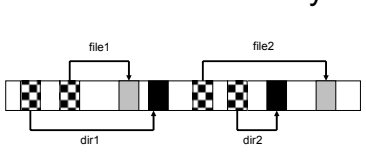
LFS – основной подход

- Весь жесткий диск представляет собой *журнал*
- Журнал – структура данных, запись в которую производится только в конец
- Если жесткий диск обслуживается как журнал, практически не будет операций поиска
- Любой файл всегда увеличивается последовательно
- Новые данные и метаданные (i-nodes, каталоги) накапливаются в буферном кэше, и впоследствии записываются вместе большими блоками (например, сегментами по 0,5 Мб или 1 Мб)
- Такой подход значительно повышает производительность дисковых операций

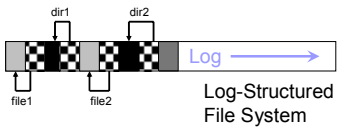
Итак, вы можете улучшить ситуацию, но…

- Если кэш становится большим, большинство операций чтения выполняются из него
- Вне зависимости от того, как вы кэшируется операции записи – *в конечном итоге* данные будут записаны на диск
- Таким образом, в трафике дисковых операций большинство будут составлять операции записи
- Если вы в конце концов помещаете блоки (i-node или блоки данных) назад в то же самое место на диске, откуда они были считаны ранее, то даже при использовании изощренного алгоритма переупорядочивания очереди запросов необходимо будет выполнять множество перемещений головок диска (которые составляют большую часть стоимости дисковых операций). То есть, вы просто не сможете использовать диск эффективно.

LFS vs. UNIX File System или FFS



Unix File System



Log-Structured File System

Блоки, записываемые при создании 2 файлов размером по 1 блоку: dir1/file1 и dir2/file2 в UFS и LFS

LFS – перспективная идея

- Предположим следующее: все, что вы записываете на диск – это журнал изменений, сделанных в файлах
  - журнал содержит измененные блоки данных и измененные блоки метаданных
  - в памяти отводится под буфер большой участок памяти ("сегмент") – 512 Кб или 1 Мб
  - по заполнении, буфер записывается на диск одной большой и эффективной операцией
    - сразу получаем уменьшение операций поиска в 1Мб/4Кб = 250 раз
- Таким образом, диск содержит один большой журнал изменений, содержащий записываемые потоком сегменты

Проблемы LFS

- Поиск данных, записанных в журнал
  - FFS размещает файлы в хорошо известном месте, LFS записывает данные "в конец журнала"
- Поиск i-nodes!
  - в LFS i-nodes тоже помещаются в журнал!
- Управление свободным пространством диска
  - диск имеет ограниченный размер → размер журнала тоже ограничен
  - таким образом, мы не можем добавлять записи в журнал до бесконечности
    - нужно освобождать и собирать для повторного использования неиспользуемые блоки в старой части журнала
    - нужно заполнять "окна" из неиспользуемых блоков, возникающие по мере их освобождения
- Замечание: Операции чтения выполняются так же, как и в FS/FFS после нахождения i-node, операции записи – намного быстрее

Определение местоположения данных и i-nodes

- LFS использует i-nodes для описания размещения данных, как и FS/FFS
- LFS добавляет i-nodes в конец журнала, как и данные
  - при таком подходе i-node трудно найти
- Решение
  - используем дополнительный уровень косвенной адресации – "карты i-node" (i-node maps)
    - карта i-node отображает номера файлов (i-node numbers) на адрес i-node
  - адреса карт i-node располагаются в специальных регионах контрольных точек
    - регионы контрольных точек имеют фиксированное расположение
  - в целях повышения производительности, карты i-node кэшируются в памяти

Очистка сегментов – подробности

- Основной проблемой при использовании LFS является очистка сегментов, то есть формирование блоков непрерывного свободного пространства
- Демон очистки сегментов "очищает" старые сегменты, то есть берет несколько малозаполненных сегментов и перепаковывает их содержимое, формируя полностью заполненный сегмент + свободное пространство
- Демон очистки выбирает сегменты на диске, основываясь на следующих характеристиках:
  - степень использования: сколько свободного пространства можно получить, очистив сегмент
  - возраст: насколько вероятно изменение сегмента в ближайшем будущем

Управление свободным пространством

- Операции чтения такие же, как в UNIX ФС и FFS (после того, как мы нашли i-node)
  - карта i-node кэшируется в память, с ее помощью находим i-node, в i-node описывается размещение блоков данных
- Каждая операция записи вызывает добавление новых блоков к текущему "буферному сегменту" в памяти
  - когда "сегмент" заполняется, он записывается на диск
- Со временем, сегменты в журнале становятся фрагментированными, поскольку мы замещаем старые блоки в файлах новыми
  - мы можем провести "сборку мусора"
    - выбираем сегменты с небольшим количеством актуальных данных
    - переразмещаем актуальные данные
    - можно использовать освободившиеся сегменты

LFS: последствия сбоев

- Что произойдет в случае сбоя?
  - вы потеряете часть информации
  - но журнал на диске представляет собой корректную версию файловой системы на некоторый момент времени
- Предположим, вы выполняете операцию чтения из файла
  - как только вы нашли его текущую версию i-node – все отлично
  - карты i-node обеспечивают дополнительный уровень косвенной адресации, делающий это возможным
    - мы не будем рассматривать конкретные детали

Очистка сегментов

- Журнал разбивается на сегменты большого размера
- Сегменты разбросаны по диску и организованы в список
  - сегменты могут быть расположены где угодно на диске
- Очищая сегменты, освобождаем пространство
  - считываем сегмент
  - копируем актуальные данные в конец журнала
  - теперь у нас еще один свободный сегмент
- Очистка сегментов является серьезной проблемой
  - имеет большую стоимость
  - когда эффективнее делать очистку?

- Как избежать переполнения диска (ведь наш журнал просто растет)?
  - демон очистки сегментов объединяет актуальные данные из нескольких старых сегментов в новый сегмент, очищая старые сегменты для повторного использования
    - здесь мы также не будем рассматривать детали

LFS vs. FFS

- LFS лучше FFS
  - большой объем работы с метаданными
    - при записи малых объемов данных в файлы
    - при удалениях
  - (метаданные требуют дополнительной операции записи, а FFS выполняет их синхронно)
- LFS хуже FFS
  - файлы могут перезаписываться частично по произвольным смещениям и в произвольном порядке
    - в результате их данные будут разбросаны по всему журналу

LFS: история

- Создатели – Мендель Розенблюм (Mendel Rosenblum) и его наставник Джон Остерхаут (John Ousterhout), Berkeley, 1991
  - Розенблюм позже стал профессором Стэнфордского университета и сооснователем VMWare
- Марго Сельтцер (Margo Seltzer) – бывшая студентка Беркли, впоследствии преподаватель в Гарварде – в 1995 году опубликовала статью, в которой сравнивались LFS и традиционная FFS и отмечалась низкая производительность LFS при определенных условиях, встречающихся в реальных задачах
- Остерхаут опубликовал статью "Critique of Seltzer's LFS Measurements", опровергающую ее аргументы
- Сельтцер опубликовала "A Response to Ousterhout's Critique of LFS Measurements", опровергающую опровержение
- Остерхаут опубликовал "A Response to Seltzer's Response", опровергающее опровержение опровержения

- Мораль
  - если вы собираетесь заняться исследованиями в области ОС, сначала запаситесь толстой кожей
  - *Очень* сложно предсказать условия, в которых будет использоваться файловая система
    - Соответственно, тяжело разработать даже адекватные тесты производительности – что же говорить о структуре файловой системы
  - Очень сложно измерить производительность файловой системы при ее практическом использовании
    - необходимо рассматривать ОГРОМНОЕ число параметров, включая особенности нагрузки и аппаратной архитектуры