1. **Модель непротиворечивости - потенциальная непротиворечивости.**

Репликация данных

Важным вопросом для распределенных систем является репликация данных. Данные обычно реплицируются для повышения надежности и увеличения производительности. Одна из основных проблем при этом — сохранение непротиворечивости реплик. Говоря менее формально, это означает, что если в одну из копий вносятся изменения, то нам необходимо обеспечить, чтобы эти изменения были внесены и в другие копии, иначе реплики больше не будут одинаковыми.

Вопросы, связанные с репликацией

1. как должны расходиться обновления по копиям
2. поддержка непротиворечивости
3. проблема кеширования

Основные проблемы репликации

Первая проблема - **обеспечение взаимного исключения** при доступе к объекту. Существует два решения:

- объект защищает себя сам (к примеру, доступ к объекту совершается при помощи синхронизированных методов в Java - каждому клиенту будет создано по потоку и виртуальная Java-машина не даст воспользоваться методом доступа к ресурсу обоим потокам в один момент времени)
- система защищает объект (ОС сервера создает некоторый адаптер, который делает доступным объект, только одному клиенту в один момент времени)

Вторая проблема - **поддержка непротиворечивости реплик**. Здесь снова выделяют два пути решения:

- решение основанное на осведомленности объекта о том, что он был реплицирован. По сути, обязанность поддерживать непротиворечивость реплик перекладывается на сам объект. То есть в системе нет централизованного механизма поддержки непротиворечивости репликаций. Преимущество в том, что объект может реализовывать некоторые специфичные для него методы поддержки непротиворечивости.
- обязанность поддержки непротиворечивости накладывается на систему управления распределенной системой. Это упрощает создание реплицируемых объектов, но если для поддержки непротиворечивости нужны некоторые специфичные для объекта методы, это создает трудности.

Модели непротиворечивости, ориентированные на данные

По традиции непротиворечивость всегда обсуждается в контексте операций чтения и записи над совместно используемыми данными, доступными в распределенной памяти (разделяемой) или в файловой системе (распределенной).

Модель непротиворечивости (consistency model), по существу, представляет собой контракт между процессами и хранилищем данных. Он гласит, что если процессы согласны соблюдать некоторые правила, хранилище соглашается работать правильно. То есть, если процесс соблюдает некоторые правила, он может быть уверен, что данные которые он читает являются актуальными. Чем сложнее правила - тем сложнее их соблюдать процессу, но тем с большей вероятностью прочитанные данные действительно являются актуальными.

- **Строгая**

Наиболее жесткая модель непротиворечивости называется строгой непротиворечивостью (strict consistency). Она определяется следующим условием: *всякое чтение элемента данных x возвращает значение, соответствующим результату последней записи X.*

Это определение естественно и очевидно, хотя косвенным образом подразумевает существование абсолютного глобального времени (как в классической физике), в котором определение «последней» однозначно. Однопроцессорные системы традиционно соблюдают строгую непротиворечивость, и программисты таких систем склонны рассматривать такое поведение, как естественное. Рассмотрим следующую программу:

a = 1; a = 2; print(a);

Система, в которой эта программа напечатает 1 или любое другое значение, кроме 2, быстро приведет к появлению толпы возбужденных программистов и массе полезных мыслей.

Основная идея: это идеальный случай. То есть система параллельная, но все процессы видят последовательность записей в общий ресурс, в таком порядке, как если бы все процессы выполнялись на однопроцессорной машине.

- **последовательная**

Последовательная непротиворечивость (sequential consistency) — это менее строгая модель непротиворечивости. В общем, хранилище данных считается последовательно непротиворечивым, если оно удовлетворяет следующему условию: *результат любого действия такой же, как если бы операции (чтения и записи) всех процессов в хранилище данных выполнялись бы в некотором последовательном порядке, причем операции каждого отдельного процесса выполнялись бы в порядке, определяемом его программой.*

Это определение означает, что когда процессы выполняются параллельно на различных (возможно) машинах, любое правильное чередование операций чтения и записи является допустимым, но все процессы видят одно и то же чередование операций. Отметим, что о времени никто не вспоминает, то есть никто не ссылается на «самую последнюю» операцию записи объекта. Отметим, что в данном контексте процесс «видит» записи всех процессов, но только свои собственные чтения.

Основная идея: все процессы видят записи в общий ресурс (от любых процессов) в одинаковом порядке, хотя этот порядок может меняться (потому, что вариантов последовательности записей от разных процессов может быть много).

- **причинная**

Модель причинной непротиворечивости (causal consistency) представляет собой ослабленный вариант последовательной непротиворечивости, при которой проводится разделение между событиями, потенциально обладающими причинно-следственной связью, и событиями, ею не

обладающими. Если событие B вызвано предшествующим событием A или находится под его влиянием, то причинно-следственная связь требует, чтобы все окружающие наблюдали сначала событие A, а затем B.

Для того чтобы хранилище данных поддерживало причинную непротиворечивость, оно должно удовлетворять следующему условию: *операции записи, которые потенциально связаны причинно-следственной связью, должны наблюдаться всеми процессами в одинаковом порядке, а параллельные операции записи могут наблюдаться на разных машинах в разном порядке.*

Основная идея: Последовательность выполнения задач, которые связанны между собой (к примеру, одна задача пользуется результатами другой) должны быть одинаково видна всем процессам, причем задачи в ней должны идти в последовательности, которую налагает на них зависимость (то есть если сначала выполняется А и его результат передается Б, то все процессы видят последовательность А Б)

- **непротиворечивость FIFO**

В случае причинно-следственной непротиворечивости допустимо, чтобы на различных машинах параллельные операции записи наблюдались в разном порядке, но операции записи, связанные причинно-следственной связью, должны иметь одинаковый порядок выполнения, с какой бы машины ни велось наблюдение. Следующим шагом в ослаблении непротиворечивости будет освобождение от последнего требования. Это приведет нас к непротиворечивости FIFO (FIFO consistency), которая подчиняется следующему условию: *операции записи, осуществляемые единичным процессом, наблюдаются всеми остальными процессами в том порядке, в котором они осуществляются, но операции записи, происходящие в различных процессах, могут наблюдаться разными процессами в разном порядке.*

Основная идея: Еще более ослабленный вариант причинной непротиворечивости. В данном случае, порядок в последовательности должен сохранятся только в том случае, если операции выполняются на одном процессоре.

- **слабая**

Рассмотрим случай, когда процесс внутри критической области заносит записи в реплицируемую базу данных. Хотя другие процессы даже не предполагают работать с новыми записями до выхода этого процесса из критической области, система управления базой данных не имеет никаких средств, чтобы узнать, находится процесс в критической области или нет, и может распространить изменения на все копии базы данных.

Наилучшим решением в этом случае было бы позволить процессу покинуть критическую область и затем убедиться, что окончательные результаты разосланы туда, куда нужно, и не обращать внимания на то, разосланы всем копиям промежуточные результаты или нет. Это можно сделать, введя так называемую переменную синхронизации (synchronization variable). Переменная синхронизации S имеет только одну ассоциированную с ней операцию, synchronize(S), которая синхронизирует все локальные копии хранилища данных. Напомним, что процесс P осуществляет операции только с локальной копией хранилища. В ходе синхронизации хранилища данных все локальные операции записи процесса P распространяются на остальные копии, а операции записи других процессов — на копию данных P.

Используя переменные синхронизации для частичного определения непротиворечивости, мы приходим к так называемой слабой непротиворечивости (weak consistency).

Основная идея: Если процесс активно работает с репликой, ему, скорее всего, не нужно что бы промежуточные результаты его работы попадали в общий доступ. Поэтому он сначала проводит несколько операций с репликой (блок операций), а после этого вызывает процедуру синхронизации, которая делает актуальной все реплики.

- **свободная**

Слабая непротиворечивость имеет проблему следующего рода: когда осуществляется доступ к переменной синхронизации, хранилище данных не знает, то ли это происходит потому, что процесс закончил запись совместно используемых данных, то ли наоборот начал чтение данных. Соответственно, оно может предпринять действия, необходимые в обоих случаях, например, убедиться, что завершены (то есть распространены на все копии) все локально инициированные операции записи и что учтены все операции записи с других копий. Если хранилище должно распознавать разницу между входом в критическую область и выходом из нее, может потребоваться более эффективная реализация. Для предоставления этой информации необходимо два типа переменных или два типа операций синхронизации, а не один.

Свободная непротиворечивость (release consistency) предоставляет эти два типа. Операция захвата (acquire) используется для сообщения хранилищу данных о входе в критическую область, а операция освобождения (release) говорит о том, что критическая область была покинута. Эти операции могут быть реализованы одним из двух способов: во-первых, обычными операциями над специальными переменными; во-вторых, специальными операциями. В любом случае программист отвечает за вставку в программу соответствующего дополнительного кода, реализующего, например, вызов библиотечных процедур acquire и release или процедур enter_critical_region и leave_critical_region.

Основная идея: Это модификация слабой непротиворечивости, но теперь есть не одна процедура “синхронизировать”, а две процедуры “вход в критическую секцию” (надо убедится что у меня актуальные данные), “выход из критической секции” (надо разослать всем то, что я понаделал и узнать что понаделали другие). Так же вводятся барьеры.

- **ленивая**

У свободной непротиворечивости имеется такая реализация, как ленивая свободная непротиворечивость (lazy release consistency). При ленивой свободной непротиворечивости в момент освобождения ничего никуда не рассылается. Взамен этого в момент захвата процесс, пытающийся произвести захват, должен получить наиболее свежие данные из процесса или процессов, в которых они хранятся. Для определения того, что эти элементы данных действительно были переданы, используется протокол отметок времени.

Основная идея: Когда процесс выходит из критической секции он не рассылает всем обновления, которые он произвел. Поэтому другие процессы, при входе в критическую секцию должны сами запрашивать у всех остальных изменения.

- **поэлементная**

Еще одна модель непротиворечивости, созданная для применения в критических областях, — поэлементная непротиворечивость (entry consistency). Как и оба варианта свободной непротиворечивости, она требует от программиста (или компилятора) вставки кода для захвата и освобождения в начале и конце критической области. Однако в отличие от свободной непротиворечивости, поэлементная непротиворечивость дополнительно требует, чтобы каждый отдельный элемент совместно используемых данных был ассоциирован с переменной синхронизации — блокировкой или барьером. Если необходимо, чтобы к элементам массива имелся независимый параллельный доступ, то различные элементы массива должны быть ассоциированы с различными блокировками. Когда происходит захват переменной синхронизации, непротиворечивыми становятся только те данные, которые ассоциированы с этой переменной синхронизации. Поэлементная непротиворечивость отличается от ленивой свободной непротиворечивости тем, что в последней отсутствует связь между совместно используемыми элементами данных и блокировками или барьерами, потому при захвате необходимые переменные определяются эмпирически.

Модели непротиворечивости, ориентированные на клиента

Модели непротиворечивости, описанные в предыдущем разделе, ориентированы на создание непротиворечивого представления хранилища данных. При этом делалось важное предположение о том, что параллельные процессы одновременно изменяют данные в хранилище и необходимо сохранить непротиворечивость хранилища в условиях этой параллельности. Так, например, в случае поэлементной непротиворечивости на базе объектов хранилище данных гарантирует, что при обращении к объекту процесс получит копию объекта, отражающую все произошедшие с ним изменения, в том числе и сделанные другими процессами. В ходе обращения гарантируется также, что нам не помешает никакой другой объект, то есть обратившемуся процессу будет предоставлен защищенный механизмом взаимного исключения доступ.

Возможность осуществлять параллельные операции над совместно используемыми данными в условиях последовательной непротиворечивости является базовой для распределенных систем. По причине невысокой производительности последовательная непротиворечивость может гарантироваться только в случае использования механизмов синхронизации — транзакций или блокировок.

Далее мы рассмотрим специальный класс распределенных хранилищ данных, которые характеризуются отсутствием одновременных изменений или легкостью их разрешения в том случае, если они все-таки случатся. Большая часть операций подразумевают чтение данных. Подобные хранилища данных соответствуют очень слабой модели непротиворечивости, которая носит название потенциальной непротиворечивости. После введения специальных моделей непротиворечивости, ориентированных на клиента, оказывается, что множество нарушений непротиворечивости можно относительно просто скрыть.

- **потенциальная непротиворечивость**

Степень, в которой процессы действительно работают параллельно, и степень, в которой действительно должна быть гарантирована непротиворечивость, могут быть разными. Существует множество случаев, когда параллельность нужна лишь в урезанном виде. Так, например, во многих системах управления базами данных большинство процессов не производит изменения данных, ограничиваясь лишь операциями чтения. Изменением данных занимается лишь один, в крайнем случае несколько процессов. Вопрос состоит в том, как быстро эти изменения могут стать доступными процессам, занимающимся только чтением данных. В качестве примера можно привести случай распределенных и реплицируемых баз данных (крупных), нечувствительных к относительно высокой степени нарушения непротиворечивости. Обычно в них длительное время не происходит изменения данных, и все реплики постепенно становятся непротиворечивыми. Такая форма непротиворечивости называется *потенциальной непротиворечивостью (eventual consistency).*

Потенциально непротиворечивые хранилища данных имеют следующее свойство: *в отсутствие изменений все реплики постепенно становятся идентичными.*

Потенциальная непротиворечивость, в сущности, требует только, чтобы изменения гарантированно расходились по всем репликам. Конфликты двойной записи часто относительно легко разрешаются, если предположить, что вносить изменения может лишь небольшая группа процессов. Поэтому реализация потенциальной непротиворечивости часто весьма дешева.

Основная идея: Если есть большая база данных со множеством реплик, но все в основном читают из нее, а пишет лишь ограниченное количество потоков, то есть изменения вносятся не очень часто. Потенциальная непротиворечивость гарантирует, что если базу никто не будет трогать на запись какое-то время, то в конце концов все реплики станут актуальными.

- **монотонное чтение**

Хранилище данных обеспечивает непротиворечивость монотонного чтения (monotonic-read consistency), если удовлетворяет следующему условию: *если процесс читает значение элемента данных x, любая последующая операция чтения x всегда возвращает то же самое или более позднее значение.* Другими словами, непротиворечивость монотонного чтения гарантирует, что если процесс в момент времени t видит некое значение x, то позже он никогда не увидит более старого значения x.

Основная идея: Если процесс один раз прочитал значение, то при повторном считывании этого значения (даже из другой реплики) ему будет возвращено это же значение, или более позднее значение (которое реально появилось позже, чем первоначально считанное значение).

- **монотонная запись**

Во многих ситуациях важно, чтобы по всем копиям хранилища данных в правильном порядке распространялись операции записи. Это можно осуществить при условии непротиворечивости монотонной записи (monotonic-xmite consistency). Хранилище должно выполнять следующее условие: *операция записи процесса в элемент данных x завершается раньше любой из последующих операций записи этого процесса в элемент x.*

Здесь завершение операции записи означает, что копия, над которой выполняется следующая операция, отражает эффект предыдущей операции записи, произведенной тем же процессом, и при этом не имеет значения, где эта операция была инициирована. Другими словами, операция записи в копию элемента данных X выполняется только в том случае, если эта копия соответствует результатам предыдущей операции записи, выполненных над другими копиями x.

Основная идея: Процесс может совершить операцию записи в реплику, только в том случае если все предыдущие операции записи (возможно от других потоков и совершенные над другими репликами) уже применены к данной реплике.

- **чтение собственных записей**

Хранилище данных обладает свойством непротиворечивости чтения собственных записей (read-your-writes consistency), если оно удовлетворяет следующему условию: *результат операций записи процесса в элемент данных x всегда виден последующим операциям чтения x этого же процесса.*

Другими словами, операция записи всегда завершается раньше следующей операции чтения того же процесса, где бы ни происходила эта операция чтения.

Основная идея: Если процесс записал что-то в реплику, для него эти изменения произойдут моментально и он сразу же сможет ими воспользоваться. (Пример: мы меняем пароль, новый пароль отправляется на основной сервер, там хешируется и рассылается репликам. Пока до локальной реплики не дойдет новый хеш, процесс использующий эту реплику не сможет использовать новый пароль. Чтение собственных записей подразумевает, что локальную реплику мы обновим немедленно, не дожидаясь пока до нее дойдет обновление от основного сервера)

- **запись за чтением**

Модель, в которой изменения распространяются как результаты предыдущей операции чтения. В этом случае говорят, что хранилище данных обеспечивает непротиворечивость записи за чтением (writes-follow-reads consistency), если соблюдается следующее условие: *операция записи в элемент данных x процесса, следующая за операцией чтения x того же процесса, гарантирует, что будет выполняться над тем же самым или более свежим значением x, которое было прочитано предыдущей операцией.*

Иными словами, любая последующая операция записи в элемент данных x, производимая процессом, будет осуществляться с копией x, которая имеет последнее считанное тем же процессом значение x.

Пример: Непротиворечивость записи за чтением позволяет гарантировать, что пользователи сетевой группы новостей увидят письма с ответами на некое письмо только позже оригинального письма. Поясним проблему. Представьте себе, что пользователь сначала читает письмо A. Затем он

реагирует на него, посылая письмо B. Согласно требованию непротиворечивости записи за чтением, письмо B будет разослано во все копии группы новостей только после того, как туда будет послано письмо A.

2. Особенности решения задачи хранения места расположения файла в различных ОС.

Файл — законченная именованная совокупность информации, набор данных, используемый программой, или документ, созданный пользователем. **Файл** — основной элемент хранения данных в компьютере; такая организация позволяет отличить один набор данных от другого. Файл является единым связным элементом, который пользователь может найти, изменить, удалить, сохранить или послать на устройство вывода.

Особенности размещения файлов в HPFS С точки зрения размещения файлы, каталоги и их расширенные атрибуты (если они не помещаются в FNode) рассматриваются HPFS как наборы экстентов. Экстент — это кусок файла лежащий в последовательных секторах. Каждый экстент описывается двумя числами: номером первого сектора и длиной (в секторах). Два последовательных экстента всегда объединяются HPFS в один. Минимальный размер экстента — один сектор. Так как расстояние между соседними битмапами свободных секторов равно 16MB, то и размер максимального экстента равен 16MB. Если файл состоит из восьми или менее экстентов, то его описание целиком хранится в FNode.

Если файл состоит более чем из восьми экстентов, то его описание может занимать несколько секторов расположенных поближе к файлу, при этом эти сектора содержат не список, а прошитое сбалансированное дерево экстентов (B+-Tree). Дерево построено так, что его разбалансировка никогда не превышает 1/3 по объему, и оно не отличается от оптимального более чем на один уровень. Корень дерева находится в FNode, причем может содержать до 12 элементов. Каждый дополнительный сектор представляющий собой ветку дерева содержит до 60 элементов, а лист — 40 элементов. Таким образом, если файл состоит из экстентов по одному сектору (этого никогда не будет!) и имеет размер 2GB, для его описания потребуется дерево следующей структуры: 12*60*60*60*40=53MB листьев и 1.7MB веток. Для случайном доступа к любой части файла при этом потребуется (в худшем случае) 5 операций чтения управляющих структур.Реальные файлы состоят из 1-3 экстентов.
Индексные Дескрипторы — *это объект Unix, который ставится во взаимнооднозначное соответствие с содержимым файла.* То есть для каждого ИД существует только одно содержимое и наоборот, за исключением лишь той ситуации, когда файл ассоциирован с каким-либо внешним устройством. Напомним содержимое ИД:

- поле, определяющее тип файла (каталоги и все остальные файлы);
- код привилегии/защиты;
- количество ссылок к данному ИД из всевозможных каталогов файловой системы;
- (нулевое значение означает свободу ИД)
- длина файла в байтах;
- даты и времена (время последней записи, дата создания и т.д.);
- поле адресации блоков файла.

Как видно — в ИД нет имени файла. Давайте посмотрим, как организована адресация блоков, в которых размещается файл.
В поле адресации находятся номера первых десяти блоков файла, то есть если файл небольшой, то вся информация о размещении данных файла находится непосредственно в ИД. Если файл превышает десять блоков, то начинает работать некая списочная структура, а именно, 11й элемент поля адресации содержит номер блока из пространства блоков файлов, в которых размещены 128 ссылок на блоки данного файла. В том случае, если файл еще больше — то используется 12й элемент поля адресации. Сутьего в следующем — он содержит номер блока, в котором содержится 128 записей о номерах блоках, где каждый блок содержит 128 номеров блоков файловой системы. А если файл еще больше, то используется 13 элемент — где глубина вложенности списка увеличена еще на единицу.
Таким образом мы можем получить файл размером $(10+128+128^2+128^3)*512$.
Если мы зададим вопрос — зачем все это надо (таблицы свободных блоков, ИД и т.д.), то вспомним, что мы рассматриваем взаимосвязь между аппаратными и программными средствами вычислительной системы, а в данном случае подобное устройство файловой системы позволяет сильно сократить количество реальных обменов с ВЗУ, причем эшелонированная буферизация в ОС Unix делает число этих обменов еще меньше

3. Последовательность операций поиска места расположения файла В UNIX.

Файловая система организована в виде дерева с одной исходной вершиной, которая называется корнем (записывается: «/»); каждая вершина в древовидной структуре файловой системы, кроме листьев, является каталогом файлов, а файлы, соответствующие дочерним вершинам, являются либо каталогами, либо обычными файлами, либо файлами устройств. Имени файла предшествует указание пути поиска, который описывает место расположения файла в иерархической структуре файловой системы. Имя пути поиска состоит из компонент, разделенных между собой наклонной чертой (/); каждая компонента представляет собой набор символов, составляющих имя вершины (файла), которое является уникальным для каталога (предыдущей компоненты), в котором оно содержится. Полное имя пути поиска начинается с указания наклонной черты и идентифицирует файл (вершину), поиск которого ведется от корневой вершины дерева файловой системы с обходом тех ветвей дерева файлов, которые соответствуют именам отдельных компонент. Так, пути «/etc/passwd», «/bin/who» и «/usr/src/cmd/who.c» указывают на файлы, являющиеся вершинами дерева, изображенного на Рисунке 1.2, а пути «/bin/passwd» и «/usr/src/date.c» содержат неверный маршрут. Имя пути поиска необязательно должно начинаться с корня, в нем следует указывать маршрут относительно текущего для выполняемого процесса каталога, при этом предыдущие символы «наклонная черта» в имени пути опускаются. Так, например, если мы находимся в каталоге «/dev», то путь «tty01» указывает файл, полное имя пути поиска для которого «/dev/tty01».

Программы, выполняемые под управлением системы UNIX, не содержат никакой информации относительно внутреннего формата, в котором ядро хранит файлы данных, данные в программах представляются как бесформатный поток байтов. Программы могут интерпретировать поток байтов по своему желанию, при этом любая интерпретация никак не будет связана с фактическим способом хранения данных в операционной системе. Так, синтаксические правила, определяющие задание метода доступа к данным в файле, устанавливаются системой и являются едиными для всех программ, однако семантика данных определяется конкретной программой. Например, программа форматирования текста troff ищет в конце каждой строки текста символы перехода на новую строку, а программа учета системных ресурсов acctcom работает с записями фиксированной длины. Обе программы пользуются одними и теми же системными средствами для осуществления доступа к данным в файле как к потоку байтов, и внутри себя преобразуют этот поток по соответствующему формату. Если любая из программ обнаружит, что формат данных неверен, она принимает соответствующие меры.

Каталоги похожи на обычные файлы в одном отношении; система представляет информацию в каталоге набором байтов, но эта информация включает в себя имена файлов в каталоге в объявленном формате для того, чтобы операционная система и программы, такие как ls (выводит список имен и атрибутов файлов), могли их обнаружить.

Права доступа к файлу регулируются установкой специальных битов разрешения доступа, связанных с файлом. Устанавливая биты разрешения доступа, можно независимо управлять выдачей разрешений на чтение, запись и выполнение для трех категорий пользователей:

владельца файла, группового пользователя и прочих. Пользователи могут создавать файлы, если разрешен доступ к каталогу. Вновь созданные файлы становятся листьями в древовидной структуре файловой системы.

Блок начальной загрузки	Суперблок	Индексные дескрипторы	Блоки файлов	Область сохранения
0				N-M+1

Суперблок файловой системы - содержит оперативную информацию о состоянии файловой системы, а также данные о параметрах настройки файловой системы. В частности суперблок имеет информацию о

- количестве индексных дескрипторов (ИД) в файловой системе;
- размере файловой системы;
- свободных блоках файлов;
- свободных ИД;
- еще ряд данных, которые мы не будем перечислять в силу уникальности их назначения

Вот информация о суперблоке. Какие можно сделать выводы и замечания?

- суперблок всегда находится в ОЗУ; (Недостаток)
- все операции по освобождению блоков, занятию блоков файлов, по занятию и освобождению ИД происходят в ОЗУ (минимизация обменов с диском). Если же содержимое суперблока не записать на диск и выключить питание, то возникнут проблемы (несоответствие реального состояния файловой системы и содержимого суперблока). Но это уже требование к надежности аппаратуры системы.

Свойство файловой системы по оптимизации доступа, критерием которого является количество обменов, которые файловая система производит для своих нужд, не связанных с чтением или записью информации файлов. (Преимущество)

4. **Методы повышения эффективности управления памятью.**

Чтобы обеспечить эффективный контроль использования памяти, ОС должна выполнять следующие функции:

- -отображение адресного пространства процесса на конкретные области физической памяти;
- -распределение памяти между конкурирующими процессами;
- -контроль доступа к адресным пространствам процессов;
- -выгрузка процессов (целиком или частично) во внешнюю память, когда в оперативной памяти недостаточно места;
- -учет свободной и занятой памяти.

Проблема, присущая MS-DOS — это ограниченный объем оперативной памяти, к которой могут обращаться программы — 640 Кбайт. Эти ограничения вызваны архитектурой микропроцессоров 8088 и 8086, используемых в IBM PC/XT: эти микропроцессоры могут адресовать только 1 Мбайт памяти. В дальнейшем объем доступной оперативной памяти еще уменьшается, так как адреса между 640 Кбайт и 1 Мбайт необходимы для работы видео — адаптера и для BIOS ПЗУ. Процессоры i80286, i80386, i80486, Pentium позволяют адресовать гораздо больший объем памяти, но MS-DOS не способна использовать память более 640Кб.

Были попытки обойти это ограничение MS-DOS на объем доступной оперативной памяти, такие как стандарты памяти EMS и EEMS и их расширения. Они открывают подобие "окон" в дополнительной памяти, но это процесс сложный и неэффективный.

OS/2 ликвидирует 640-килобайтное ограничение памяти. Когда OS/2 управляет памятью программы, данные могут в действительности превышать объем физически доступной памяти.

Программы делятся на сегменты, и, если не хватает пространства в ОЗУ для полной программы, OS/2 "сбрасывает" некоторые сегменты из памяти на диск, где они остаются, пока не понадобятся. Во многих случаях программист может игнорировать предел объема физической памяти в машине.

Когда программа запущена, операционная система заботится о переносе соответствующих сегментов в и из памяти. Когда не хватает памяти для всех одновременно работающих задач, система сохраняет на диске те сегменты, к которым не было обращений в течение долгого времени. Это называется алгоритм LRU (по "самому давнему использованию").

Аппаратура процессоров 80386 и выше переводит ссылки на виртуальные адреса в физические адреса. Каждый выполнимый процесс снабжается таблицей, которая содержит информацию об используемых сегментах, такую как их расположение и длина. Операционная система изменяет эти таблицы, чтобы описать загружаемый процесс, а аппаратура использует их для перевода ссылок на виртуальные адреса в адреса физической памяти. Если необходимого участка нет в памяти, соответствующий сегмент загружается с диска в память.

Эта деятельность незаметна для программиста, которому (обычно) не нужно знать, находится ли данный сегмент в памяти или на диске.

В языке Си компилятор создает сегменты, необходимые для кода программы, для стека и для переменных, определенных в программе. Однако, программист может создать другие сегменты данных, к которым можно обращаться независимо от программы, которые могут быть общими для различных процессов и которые могут освобождаться, когда необходимость в них отпадет. API OS/2 обеспечивает обширный набор функций для создания, использования и освобождения сегментов.

Версии OS/2 используют возможности процессора 80386 и выше, которые позволяют непосредственно адресовать 4 Гбайт физической памяти и до 64 Тбайт виртуальной памяти (1 Тбайт = 1024 Гбайт).

5. **Когерентность данных. Проблемы. Способы решения задач согласования данных.**

Когерентность данных означает, что в любой момент времени для каждого элемента данных во всей памяти узла существует только одно его значение несмотря на то, что одновременно могут существовать несколько копий элемента данных, расположенных в разных видах памяти и обрабатываемых разными процессорами. Механизм когерентности должен следить за тем, чтобы операции с одним и тем же элементом данных выполнялись на разных процессорах последовательно, удаляя, в частности, устаревшие копии. Проще говоря, проблема когерентности памяти состоит в необходимости гарантировать, что всякое считывание элемента данных возвращает последнее по времени записанное в него значение.

Разница времен доступов к данным в значительной степени влияет на распределение заданий и ресурсов. Для решения этой проблемы необходимо обеспечить согласование местонахождения данных, необходимых для выполнения процессов, и адреса вычислительного узла на котором будет выполняться процесс в соответствии с расписанием. Факторами, которыми определяются физическое согласование данных являются:

- физическое распределение данных и ресурсов, нужных для данного класса заданий;
- трудоемкость действий, выполняемых системой для выполнения задания;

необходимое время для перемещения данных и заданий в вычислительный узел

Когерентность происходит от латинского слова "cohaerens", что буквально переводится как "находящийся в связи", а в более широком смысле означает коррелированность (согласованность). Применительно к многопроцессорным системам когерентность означает, что процессоры согласуют свою работу при обращении к совместно используемым ресурсам и в первую очередь - оперативной памяти (порты ввода/вывода и дисковую подсистему мы рассматривать не будем, поскольку об этом успешно заботится операционная система).

Даже в полностью распараллеленном приложении процессоры вынуждены взаимодействовать друг с другом, обмениваясь данными (мы уже приводили простейший пример с задачей A -> B; C -> D). Вот вполне типичная ситуация. Имеется ячейка памяти X, содержащая значение A, которое считывает процессор CPU1, после чего процессор CPU2 записывает в X значение B. Допустим, что процессор CPU1 (не знающий, что содержимое X уже изменено) увеличивает его на единицу (а почему бы и нет) и записывает обратно, уничтожая тем самым результат работы процессора CPU2.

Теперь вспомним, что все современные процессоры имеют кэш-память, причем этой памяти очень много (зачастую намного больше мегабайта в пересчете на каждый процессор). Задумаемся, что произойдет, если при решении некоторой подзадачи процессор CPU1 запишет все (или часть вычисленных данных) в свой собственный кэш и тут же переключится на решение другой подзадачи, предоставляя процессору CPU2 возможность продолжить обработку данных, которую он считает из... основной оперативной памяти, содержимое которой осталось неизмененным и результатов вычислений там нет.

Конечно, кто-то может сказать: а нечего перекладывать дальнейшую обработку данных на процессор CPU2, пусть ей занимается CPU1! Все это верно, конечно. Никто же не спорит! Но... как быть, если операционная система не позволяет закреплять потоки за процессорами? И что делать, если нам, например, необходимо просуммировать N чисел на K процессорах? Да нет ничего проще! Разбиваем N чисел на K блоков, считаем сумму каждого из них на всех процессорах независимо от других (от перестановки слагаемых сумма, как известно, не меняется), после чего нам останется только сложить K чисел. Если N много больше K (а обычно так и есть), то время выполнения задачи обратно пропорционально количеству процессоров с коэффициентом пропорциональности близким к единице. Близким, а не равным, потому что пропускная способность памяти - это еще одна фундаментальная проблема, но о ней мы еще поговорим, а сейчас обратим наше внимание на то, что на последней стадии операции мы вынуждены обращаться к результатам вычислений других процессоров, а они у каждого из них с вероятностью, близкой к единице, находятся в кэш-памяти.

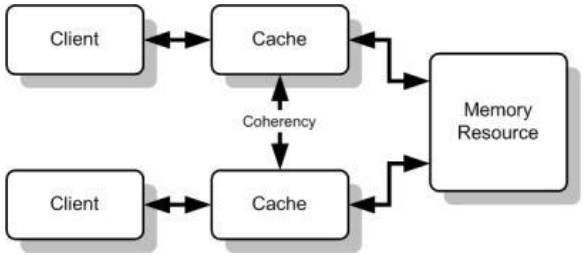
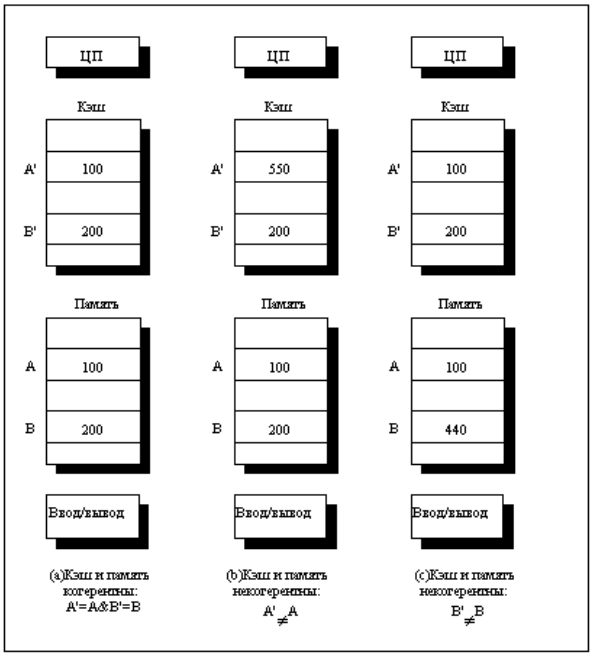


Рисунок 5. Кэш-память всех процессоров должна быть когерентна.

Существует два выхода из положения: либо вообще отказаться от кэширования, каждый раз обращаясь напрямую к оперативной памяти (да! мы даже ячейку в регистр теперь не можем поместить), либо разработать протокол, позволяющий снабжать процессоры информацией о том, какие именно ячейки находятся в кэш памяти, какие из них изменены и если процессор CPU1 обращается к ячейке памяти, которую процессор CPU2 загрузил в свой кэш и модифицировал, то CPU2 должен либо переслать модифицированную ячейку в кэш процессора CPU1, либо выгрузить кэш-строку (минимальную порцию обмена кэша с памятью) в основное ОЗУ, откуда его сможет подхватить CPU2.

Первый способ откинем сразу. Отказ от кэширования на данном этапе развития вычислительной техники невозможен, уж слишком велика разница быстродействия ядра процессора и оперативной памяти. Так что, приходится согласовывать кэш-память процессоров посредством тех или иных протоколов, задействующих системную шину, тактовая частота которой зачастую в десятки раз ниже частоты ядра и которая используется для общения процессора с "внешним миром". Короче говоря, без накладных расходов тут никак не обойтись и хотя даже на дешевых процессорах типа Pentium-III/4 помимо системной шины предусмотрены специальные выводы, разгружающие шину и берущие задачи по согласованию кэшей на себя, частота их импульсов все равно намного меньше, чем у ядра. Потому что проводники, соединяющие соседние процессоры намного длиннее, чем внутрикристальные перемычки и максимально возможная тактовая частота упирается в чисто физические ограничения, которые очень сложно преодолеть.



A' и B' - кэшированные копии элементов A и B в основной памяти

a) Когерентное состояние кэша и основной памяти.

b) Предполагается использование кэш-памяти с отложенным обратным копированием, когда ЦП записывает значение 550 в ячейку A. В результате A' содержит новое значение, а в основной памяти осталось старое значение 100. При попытке вывода A из памяти будет получено старое значение.

c) Подсистема ввода/вывода вводит в ячейку памяти B новое значение 440, а в кэш-памяти осталось старое значение B.

Рисунок 6. Иллюстрация проблемы когерентности кэш-памяти.

Протоколов для поддержки когерентности разработано много. Это и MSI (образованный по первым буквам флагов Modified/Shared/Invalid, отражающих состояние кэш-блоков), и MESI (Modified/Exclusive/Shared/Invalid), и MOSI (Modified/Owned/Shared/Invalid), и многие другие. Подробно обсуждать их достоинства и недостатки никакого смысла нет, поскольку указанные протоколы реализованы внутри процессора и являются неотъемлемой частью его архитектуры, на которую конечный потребитель воздействовать не в состоянии.

Возвращаясь к нашим баранам, продолжим гнуть перспективную партийную линию: вместо того, чтобы гонять данные между кэш-памятью разных процессоров, намного лучше (и правильнее) стремиться обрабатывать данные на том процессоре, в кэш-памяти которого они уже находятся. Многоядерные процессоры привлекательны в том смысле, что кэш второго уровня у них общий (нет никаких расходов на поддержку когерентности), а индивидуальные для каждого ядра кэши первого уровня находятся внутри кристалла и потому могут согласовывать свое содержимое с минимальными накладными расходами.

Появление гетерогенных многопроцессорных систем (содержащих два и более многоядерных процессоров) породило проблему привязки потоков к "своим" ядрам. До недавнего времени все процессоры (как физические, так и виртуальные) были полностью равноправными с точки зрения операционной системы и потому поток, начав свое выполнение на ядре A процессора CPU1 мог продолжить его где угодно: и на CPU1-A, и на CPU1-B, и на CPU2-A, и на CPU2-B - это уж как фишка ляжет. Очевидно, что первый вариант является наиболее предпочтительным. Второй вариант - немного похуже, но тоже, в общем-то, ничего. А вот два последних варианта - это смерть производительности.

Разработчики операционных систем отреагировали достаточно оперативно. Во-первых, они изменили алгоритмы планировки так, чтобы преимущество получал тот процессор, на котором поток был прерван, а во-вторых, предоставили в распоряжение программиста API-функции, позволяющие закреплять потоки за процессорами в принудительном порядке, обусловленным спецификой решаемой задачи. Естественно, алгоритмы планировки возымели действия стразу (достаточно перейти с Windows 2000 Server на Windows 2003 Server, чтобы почувствовать разницу), а вот новые API-функции требуют полного редизайна всего существующего программного обеспечения, что влечет за собой намного большие издержки, чем обновление операционной системы (что касается UNIX'а, то там вообще достаточно обновить ядро, не трогая всего остального).