|  |
| --- |
| Крюков Виктор Александрович (krukov@keldysh.ru), Бахтин Владимир Александрович (bakhtin@keldysh.ru) |
| Распределённые системы |
| Курс отличается от «распределённые операционные системы» |

|  |
| --- |
| avasite  01.01.2014 |

**Лекция 1**

На экзамене так же дают практические задачи – типо реализовать алгоритм для «а вот програмка»

Но экзамен устный (и задачи делают так, чтобы они покрывали как можно больше тем)

Разрешают всем пользоваться

Операционная система рассматривается с 2-х точек зрения:

1. Это абстрактная машина, на которой писать программы удобнее для программиста (есть некоторый уровень абстракции (например ввод/вывод))
2. Это менеджер ресурсов.

MS DOS фактически обеспечивал работу одной программы одного пользователя, за исключением мелочей, типо параллельного ввода с клавиатуры, но параллельности не было.

Ресурсы операционной системы – распределённые отдельные объекты.

История ОС:

Года будут условными.

1. 50-е – режим работы – **«Персональные ЭВМ» (ПЭВМ) (или «Пультовые ЭВМ»)**. Человеку в конкретное время дня (или ночи) выдавали машину в полное распоряжение. Были программные компоненты, как «служебные программы» (интерфейс пользователя (можно было попросить её распечатать что-нибудь (скажем дамп памяти))) и «Программы вв./выв.» (чтобы не изучать внутреннее устройство железа и его работу)

Пользователь – (программист, математик, физик).

Существовали операторы, которые работали быстрее и правильнее, но всё равно машина простаивала пока вокруг неё прыгали

1. Середина 50-х годов – **пакетная обработка**. Человеческое введение данных выделили в отдельную задачу (формирование лент или пакетов перфокарт).

Даже если одно-программный режим, то в памяти машины всё равно находится программа (аналог ОС), и эту программу нельзя портить, её нельзя затирать, менять, …

Это привело к требованиям от аппаратуры:

* 1. *защите ОП* (Оперативной памяти). Для этого в машине ввели элементарный способ – 2 регистра, которые говорят, куда можно писать, а куда - нет
  2. *прерывания* – для того, чтобы машина не зависала на выполнении пользовательской задачи (будь то вечный цикл, или деление на ноль, …), управление нужно передать в ОС
  3. *привилегированный режим* – который разграничивает права между пользователем и ОС (например, только ОС имеет право менять регистры из первого пункта), так же нужно защищать и внешнюю память, нельзя давать кому-попало писать куда попало, поэтому для того, чтобы записать что-то – нужно вызывать стандартное средство ОС для записи.
  4. *Таймер* – для предоставления возможности переключения работы с одного процесса на другой – позволяет сделать «псевдо»-программирование.

Без этих 4-х – организовать мультипрограммный режим – не обеспечить.

Мультипрограммирование позволило следить за работой программы и отлаживать её (ставить точки отладки, …).

1. Середина 60-х годов – **режим разделения времени (РРВ)** (в этом курсе почти не затрагиваем этот вопрос). РРВ – это возможность работы с программной многих пользователей одновременно.

Проблема работы с терминалом в том, что нужно существование не только инициативы от программы, но и инициатива от терминала, и поэтому сделали работу терминала через прерывание.

Этот режим появлялся, потому что очень важен диалог пользователя с машиной (исправление ошибок компиляции, отладка, …)

Как меру набранного времени берётся отношение (время вып. программы) / (время астрономическое), и в зависимости от этого отношения, пришлось

1. Страничная и сегментная организация памяти (ведь нужно как-то распределять память между пользователями).

Мотивы:

* 1. Разбить ОП на порции разной длинны, чтобы выделять процессам лишь кусочки памяти
  2. Удобство программиста – ведь если нужен длинный массив, а в ОП нету столько места подряд, то только сегментация и её маппинг помогут позволить разбить массив в реальной памяти и динамическое изменение массива (не знаешь, на сколько он потом вырастет, но если выделять из свободных кусов страниц, то проблем не будет)

Иметь много программ, между которыми мы переключаемся – дело плохое, потому что нужно перегружать много страниц (смена контекста, …) И поэтому появилась несколько очередей на обработку – те, которые должны выполняться часто и по-малому, и те, которые должны выполняться редко, но долгий интервал времени.

Если маппить одну виртуальную память, в несколько виртуальных памятей, разных редакторов, то при падении одного – другим ни горячо, ни холодно, и сам редактор делать – очень легко.

1. 70-е – многопроцессорные ЭВМ, сети и многомашинные комплексы.

Были проблемы с компиляцией – она была очень долгой, занимала больше половины времени. Казалось, что нужно делать машины с несколькими разными специализированными процессорами, чтобы ускорить нужное. Однако это не пошло, потому что пока делали это, уже другие сделали универсальный процессор, который работал в 10 раз быстрее (в общем – гонки, горячее время было). (он говорил, что на это нарвались те, кого он называл «мы»). Выбор между тем – делать специальную систему или универсальную систему – часто возникает.

(Чтобы запустить Буран – там использовался один язык для борта, а другой язык для земли, и там так было сделано, что язык понимал формально ту запись, на которой писались алгоритмы запуска другими людьми, этот язык был специализирован, но понимал чужое, - компромис между специализацией)

Процессоры вв/выв – пример специализации, которая прокатила.

Мотивы возникновения:

* 1. Специализация
  2. Эффективность (машина с несколькими процессорами - эффективнее)
  3. Надёжность (если в машине есть несколько процессоров, и один из них умер, то продолжить работу на остальных – не представляет особого труда) (+ возможно ещё, что, более специализированная аппаратура – более проста (занимается своим делом), а потому и надёжнее)

Появилась проблема распределения программ на несколько процессоров, ОСи были не готовы, потому что была проблема с тем, чтобы несколько программ одновременно могли обратиться к ОС, и можно было обслуживать параллельно нескольких, в связи с этим пришлось ось переписывать. Это привело к тому, что в первый момент времени ось посадили на отдельный процессор, и всё.

Сети – придумывались стандарты, которые в то время на аппаратуре не реализовывались, и появлялись проблемы.

Сети должны были обеспечивать вход с терминала в другую систему, и для этого ось имитировала «виртуальные терминалы». Нужно уметь вставить в пакет своё задание, и иметь возможность шариться по чужой файловой системе, …

Многомашинные комплексы позволяли выполнять общий пакет задания, но на разных машинах, но с единым вв/выв. Тут оси кардинально не усложнились.

1. 80-е – ПЭВТ (персональные ЭВМ) – пользователи были очень довольны, когда появлялся персональный компьютер, и не было уверенности, что всё, что придумано ранее – не нужно. Сама ОСь на ПЭВМ двигалась в направлении не многозадачности, а формировании графики, и другой «ерунды»

Потом стало ясно, что нужно где-то хранить большое количество данных, нужно сетевое подключение, нужны суперкомпьютеры, таким образом появились новые требования.

**Лекция 2**

В мультипроцессорных системах запись в кэш сквозная.

MPP – Massive Parallel Processing.

Многопроцессорные машины появлялись, потому что не получалось традиционным способом увеличивать производительность (увеличение тактовой частоты) (ибо сильно большое энергопотребление).

В основе производительности графического процессора состоит в том, что переключается выполнение с одной группы ядер на другую, где уже подгружен кэш. (так же можно делать и в обычных системах, - это подход сдвига выполнения, вместо сдвига контекста)

Гетерогенность – всё больше компьютеров используют ускорители – графические процессоры и процессоры intel. (фактически – это специализация)

Программист пишет на языках высокого уровня, и компилятор вместе с осью представляет для него из себя единый компьютер, а не многопроцессорный.

Недостатки РОС:

1. Тяжело писать ось, программы, языки программирования, компиляторы, …

Легко ошибиться и нарваться на deadlock или ещё что-нибудь.

1. …

Раньше думали, что будет одна операционная система для разных компов и эта система будет сливаться, и они будут выполнять функции распределённой операционной системы.

**ссылка на материалы**: [ftp.keldysh.ru/K\_student/Distributed\_Systems/progDS2014-09\_11.pdf](ftp://ftp.keldysh.ru/K_student/Distributed_Systems/progDS2014-09_11.pdf)

Существует 2 приёма позволяющих ускорить передачу сообщений в многопроцессорной системе:

1. Послать сообщение по нескольким маршрутам (часть туда и часть сюда)
2. Конвейеризация – послать сообщение k раз по кусочкам размера (сообщение/k) (было такое же на сетях – послать не весь пакет, а пакет кусочками) – это самое серьёзное улучшение

Может быть построено 2 варианта сети: широковещательная и не широковещательная. Наличие широковещания – очень полезно, потому что если нужно послать что-то всем, то это сделается за один раз, а не циклически каждому.

Если несколько процессоров обратилось к шине, то будет выбран тот, у которого меньше номер.

Нет никаких «захватить шину», …, есть только «послать» и «принять».

Коммуникационные операции могут быть неблокирующие и блокирующие.

Операция называется локальной, если нужна коммуникация, и не локальной. Коллективная – если операция требует участия нескольких.

Процессорное время он часто не считает, потому что процессор – гораздо быстрее, чем обмен по памяти.

Буферизация – мы можем класть в буфер данные на будущее, в случае, если шина пустует. Т.е. мы можем использовать send и receive – когда душе угодно, в расчёте на загрузку данных на будущее, но есть проблема с тем, что нужно затачиваться под конкретную машину. (к сожалению этот процесс вообще-то не автоматизирован)

Есть 2 проблемы – первая связанна с тем, что send ждёт подтверждения для receive (так нужно, потому что если данные отправят, а кто-то будет загружен и не сможет данные получить, то они будут сброшены у него, ибо девать их некуда), а другая проблема в том, что send можно сделать только после receive, и нельзя посылать биты дальше, сразу после получения первого бита сообщения.

Идея была в том, что для маленьких сообщений можно использовать несинхронный процесс передачи и сообщения будут класться в системный буфер, а большие сообщения отправляются всегда синхронным режимом (ждём получения receive, чтобы можно было сделать send, иначе тот не готов).

Ошибка в MPI – это «потенциальный deadlock», который может случиться в синхронном режиме.

Неблокирующие операции – не приостанавливают процесс, но возвращают ссылку на операцию, и можно либо сказать – ждём эту операцию, либо время от времени спрашиваем – не исполнилась ли там случайно нужная операция, а сами параллельно работаем.

Коллективные коммуникации – шине нужно их уметь реализовывать на основе операции точка-точка.

Операция AllToAll – обычно для транспонирования матрицы.

Глобальные операции – раньше это были операции между областями памяти (не был известен тип), и поэтому невозможно было сделать операции глобальными.

Операции часто рассчитывают на то, что данные распределены между процессорами одинаково.

**Лекция 3**

Владимир Александрович Бахтин

(кажется, он читает не по лекциям, а просто разговаривает (на хорошо))

Процессор сам по себе штука быстрая, проблема лишь в памяти.

Другая проблема – энергопотребление, при росте тактовой частоты температура растёт не линейно, а быстрее, но всё равно проблема в том, что не получается общаться с памятью.

Поэтому решили сделать несколько ядер, но уменьшить немного тактовую частоту.

Сейчас приходится решать проблему синхронизации нитей, одного узла, и нитей с разных узлов (обмен сообщениями тут надо пользовать, чтобы узлы общались)

Отдельные процессы – с каждым процессом есть много служебной информации, и у каждого процесса своя память, которая между процессами не расшаривается просто так (если только не пользовать соответствующие вызовы оси)

У нитей главная особенность – общая память, и поэтому экономится на смене контекста и на кеше, который не приходится перезагружать.

Проблема Гонок:

X = x + y x = x – y

Load r1, x

Load r2, y load r3, x

Add r1, r2 load r4, y

~~Store r1, x~~ sub r3, r4

Store r3, x

Store r1, x

Тут беда в том, что если это 2 операции, которые выполняют нити, то результат будет зависеть от того, кто за кем выполняется, что приводит к некоторым проблемам синхронизации, и к непредсказуемости поведения программы при её параллельном выполнении.

Критические интервалы – способ решения разграничения управления в нити, пока нить находится в критическом интервале – её нельзя прервать, но чтобы захватить критический интервал (процессорное время), то придётся отстоять в очереди, но тогда проблема с тем, что никто не должен ждать входа в критическую секцию бесконечно долго.

Алгоритм «Деккера» для входа и выхода из критической секции:

int turn;

int flag[2];

|  |  |  |
| --- | --- | --- |
| Программа | Enter\_region | Leave\_region |
| void proc (int i)  {  while (true)  {  <вычисления>  enter\_region(i);  <критическая секция>  leave\_region(i);  }  } | void enter\_region (int i)  {  try:  flag[i]=true;  while ( flag[(i+1)%2] == true )  {  if (turn == i)  continue;  flag[i] = false;  while (turn != i);  goto try;  }  } | void leave\_region (int i)  {  turn = (i + 1) % 2;  flag[i]=false;  } |

turn = 0;

flag[0]=flag[1]=false;

proc(0) && proc(1)

Некоторое улучшение (Алгоритм Петерсона):

void enter\_region (int i)

{

int other;

other = 1 - i;

flag[i]=true;

turn = i;

while (turn == i && flag[other] == true);

}

TSL – Test and Set Lock

TSL (r, s) это операция [r=s, s=1], главная особенность в том, что эти 2 операции выполняются неделимо.

enter\_region:

tsl reg, flag

cmp reg, #0

jnz enter\_region

ret

leave\_region:

move flag, #0

ret

Во всех случаях есть проблема – процесс, который ожидает, находится в активной фазе, т.е. он своими проверками и запросами забивает ресурсы машины.

Семафоры (by Дейкстра)

P(s)

[

if (s == 0)

<заблокировать процесс>

else

s = s-1;

]

V(s)

[

if (s == 0)

<разблокировать один из ранее заблокированных процессов>

s = s + 1;

]

Задача – есть процесс, который записывает данные, а второй их считывает, и нужно их развести в правильном порядке.

semaphore mutex = 1; // Нужно на момент записи, чтобы пока один писал или читал, то никто ничего не делал

semaphore full = 0; // Говорит, что место записи занято

semaphore empty = N; // Говорит, записали ли данные или нет

void producer ()

{

while (true)

{

int item;

produce\_item(&item);

P(empty); // Заняли слот владения информацией

P(mutex);

emter\_item(&item);

V(mutex);

V(full); // Объявили о том, что в буфер положили

}

}

void consumer(

{

while (true)

{

int item;

P(full); // убедились что в буфере что-то действительно есть

P(mutex);

leave\_item(&item);

V(mutex);

V(empty); // Сообщили, что слот хранения информации свободен

consume\_item(item);

}

}

Задача об обедающих философах.

Каждый философ должен взять вилку слева и справа, чтобы поесть. Однако, если каждый возьмёт по одной вилке слева, то все зависнут и никто не поест.

Поэтому можно решать как в сетях – подождали и опустили вилку, при неудаче. В сетях – этот подход хорошо работает.

Но лучше решать проблему deadlock через семафоры.

Задача Читателей и Писателей – нужно чтобы могло быть любое количество читателей, но всегда один писатель.

**Неправильное решение (будет небольшой недостаток):**

int rc = 0; // Количество читателей

void reader\_reading ()

{

while (true)

{

P(mutex);

rc++;

if (rc == 1)

P(reader)

V(mutex);

read\_DB();

P(mutex);

rc--;

if (rc == 0)

V(reader);

V(mutex);

reading();

}

}

void writer ()

{

while (true)

{

produce\_data();

P(reader);

write\_db();

V(reader);

}

}

Алгоритм не справедлив к писателю, потому что может случиться так, что писателя блокировать будут вечно, ибо прорва читателей, а надо было бы, чтобы писатель блокировал приход новых читак, (ну т.е. останавливал их, чтобы они подождали его).

Задание – исправить алгоритм выше.

**Лекция 4**

Правильное решение проблемы философов (так чтобы не случалось, что каждый философ возьмёт по одной вилке и будет ждать 2-ю, и так все зависнут)

philisopher(int i) // Как живёт философ

{

while (true)

{

think();

get\_fork(i);

eating();

put\_fork(i);

}

}

#define N 5 // количество философов и вилок

#define HUNGRY 0

#define EAT 1

#define THINK 2

int state[N]; // Массив вилок

semaphore S[N];

semaphore mutex = 1; // Массив для контроля доступа к state, на основании которого, везде происходит решение, чего делать

#define RIGHT (i+1)%N

#define LEFT (i+N-1)%N

test(int i)

{

if (state[i] == HUNGRY && state[LEFT] != EAT && state[RIGHT] != EAT)

{

state[i] = EAT;

V(S[i]); // Опустили семафор, и тем самым мы разрешаем тому кто вызвал эту функцию продолжить работать

}

}

get\_fork(int i)

{

P(mutex);

state[i] = HUNGRY;

test(i);

V(mutex);

P(S[i]); // Мы тут не блокируемся, если test прошёл успешно, т.е. нам будет разрешено продолжить есть, а иначе мы заблокируемся, и будем ждать, пока сосед нас разблокирует

}

put\_fork(int i)

{

P(mutex);

state[i] = THINK;

test(LEFT); // Если для левого всё хорошо, то мы опустим ему семафор и он, если был в застое, то разблокируется

test(RIGHT); // Если для правого всё хорошо, то мы опустим ему семафор и он, если был в застое, то разблокируется

V(mutex);

}

**Механизм событий**

Алгоритм последовательной верхней релаксации (неявная вычислительная схема)

Пусть дано:

float A[L1][L2];

for (i = 1; i < L1 - 1; i++)

for (j = 1; j < L2 - 1; j++)

A[i][j] = (A[i - 1][j] + A[i][j - 1] + A[i + 1]A[j] + A[i][j + 1]) / 4;

Наша задача – параллельно выполнить этот цикл, так чтобы результат не отличался от того, как его выполняют последовательно (проблема в том, что ветки цикла взаимосвязаны)

Суть будет в том, что мы попытаемся семафорами разделить, в соседних потоках обработку зависимых клеток (каждое ядро вычисляет столбик сверху вниз, по мере продвижения его вниз, можно подключать второе ядро для просчёта соседнего столбика, но нельзя допустить, чтобы 2-е обогнало 1-е) (мы собираемся начать вычисление из верхнего, левого угла и продолжать вычислять по диагонали сверху вниз, подключая всё новые столбцы для вычисления) (своего рода конвейер)

float A[L1][L2];

semaphore s[L1][L2]; // По умолчанию они все закрыты (инициализируем дальше)

for (j = 1; j < L2 - 1; j++)

V(s[0][j]);

parfor (i = 1; i < L1 - 1; i++) // parfor - означает, что каждая ветвь цикла будет выполняться отдельно на своём ядре

for (j = 1; j < L2 - 1; j++)

P(S[i-1][j]);

A[i][j] = (A[i - 1][j] + A[i][j - 1] + A[i + 1]A[j] + A[i][j + 1]) / 4;

V(S[i][j]);

Но если у нас ядер больше, чем столбцов, то, чтобы они не простаивали, то в данном случае можно разбить и второй цикл на части (т.е. запустить аналогично предыдущему рассуждению, но по строкам)

float A[L1][L2];

semaphore s[L1][L2]; // По умолчанию они все закрыты (инициализируем дальше)

for (i = 1; i < L2 - 1; i++)

V(s[i][0]);

for (j = 1; j < L2 - 1; j++)

V(s[0][j]);

parfor (i = 1; i < L1 - 1; i++)

parfor (j = 1; j < L2 - 1; j++)

P(S[i-1][j]);

P(S[i][j-1]);

A[i][j] = (A[i - 1][j] + A[i][j - 1] + A[i + 1]A[j] + A[i][j + 1]) / 4;

V(S[i][j]);

Заметим, что мы ждём процесс сверху и процесс снизу, сами мы освобождаем только один свой семафор – и это проблема, потому что ждут 2-е. Поэтому надо пользовать события:

**События:**

POST (S) – событие произошло – разблокируется сразу вся очередь процессов, которые ждут (в отличие от V, которое освобождало только один процесс)

WAIT (S) – (похоже на P) – процесс останавливается и стоит в очереди

CLEAR (S) – чтобы процесс забыл, был ли там раньше POST

float A[L1][L2];

event S[L1][L2]; // По умолчанию они все закрыты (инициализируем дальше)

for (i = 1; i < L2 - 1; i++)

POST(s[i][0]);

for (j = 1; j < L2 - 1; j++)

POST(s[0][j]);

for (i = 0; i < L1; i++)

for (j = 0; j < L2; j++)

CLEAR(S[i][j]);

parfor (i = 1; i < L1 - 1; i++)

parfor (j = 1; j < L2 - 1; j++)

WAIT(S[i-1][j]);

WAIT(S[i][j-1]);

A[i][j] = (A[i - 1][j] + A[i][j - 1] + A[i + 1]A[j] + A[i][j + 1]) / 4;

POST(S[i][j]);

Как промоделировать события через семафоры P и V:

POST – V

WAIT – P(S)V(S)

**Сообщения:**

Примерно в 78 году появилась работа, предлагающая использование не разделяемой памяти, а только сообщений.

SEND (destination, &message, m\_size)

RECEIVE ([source, ] &message, m\_size)

Посылки сообщений бывают блокирующие, неблокирующие, …

Задача – есть производитель и потребитель, если в очереди есть, что делать – то производитель работает, а потребитель ждёт, причём потребитель должен уметь послать N сообщений наперёд, не ожидая ответа (тут типо в этом суть – нужно уметь работать с очередью в N элементов (и забить помаксимуму)). (подразумевается один производитель и один потребитель, которые работают на разных ядрах, рассинхронизованно)

#define N 100

#define msize 4

typedef int message[msize];

producer()

{

message m;

int item;

while (true)

{

produce\_item(&item);

RECEIVE(consumer, &m, msize);

build\_message(&m, item);

SEND(consumer, &m, msize);

}

}

consumer()

{

message m;

int item;

for (int i = 0; i < N; i++)

SEND(producer, &m, msize);

while (true)

{

RECEIVE(producer, &m, msize);

extract\_item(&m, item);

SEND(producer, &m, msize);

consume\_item(item);

}

}

**Лекция 5**

Читай лекции самого лектора

Алгоритмы с маркерами хороши, когда процессы входят в критическую секцию с разной частотой, и тогда, если кому-то нужно несколько раз подряд войти в критическую секцию – то он сделает это, получив маркер.

DSM – распределённая общая память

**Лекция 6**

Читай лекции самого лектора.

**Лекция 7**

Распределённые файловые системы (немного теории, и реальный пример - NTFS)

С появлением сетей, хотелось бы:

Сетевая прозрачность (т.е. работа с удалёнными файлами не шибко отличается от работы с локальным файлом) и доступность (не должно быть каких-то больших проблем с доступностью к файлу (например у суперкомпьютеров есть проблема – миллионы ядер и все хотят читать из одного места жёсткого диска – ну и wtf?))

Файловый сервис – это как бы интерфейс, который система предоставляет своим пользователям.

Файловый сервер – тот, кто читает файл и отдаёт в интернет

Сервер директории – он указывает нам путь к файлу

Существует много разных систем:

AFS (какой-то университет)

NFS – корпорация sun

xFS – всё работает базируясь только на клиентах, без серверов

SFS – secure

GFS

HDFS – hadoop

Атрибуты файла:

CHANGE

TYPE

FS\_LOCATIONS – это атрибут, с помощью которого можно задать имена машин, на которых хранится копия этого файла

ACL (Access Control List) – обеспечивает контроль доступа к файлам (наличие этого атрибута не обязательно, его может и не быть)

Тут есть расширенный набор операций

1. Append
2. List\_dir
3. Read\_acl

И расширенный набор пользователей

1. Owner
2. Group
3. Anonymous
4. Batch – скрипт запущенный самим пользователем
5. network

RPC – модель удалённого доступа – сервер отвечает на наши запросы к файлу (при этом весь файл не выкачивается)

Другой подход – когда сначала выкачивается весь файл, а уже потом пользователь с ним работает

При работе с файлом очень важно кеширование.

Можно к файловой системе достучаться 2 разными способами:

1. /Server1/DuK1/file1
2. Монтирование другой файловой системы (не всей, там тоже можно монтировать только кусочек – выбрать нужный каталог) в какой-то из каталогов своей файловой системы

Обычно монтирование происходит при обращении к нужному каталогу, т.е. по умолчанию не монтируются сразу все каталоги всех пользователей при запуске системы.

1. Единая файловая система (AFS) – суть в том, что на всех машинах был корневой каталог afs, а потом шли каталоги кафедр, факультетов, …, т.е. придя на каждую машину – мы видим всю иерархию, хотя данные распределены

Именование файлов:

Обычно большинство систем использует двойное именование – это символьное имя для человека, и дескриптор, который ставится в соответствие имени этого файла, с которым файловая система и работает.

Таким образом, дескриптор в распределённой файловой система должен идентифицировать единственным образом файл во всей сети, а не только на локальном компьютере (потому нужно хранить, например, ip-адрес).

NFS3 – сервер без состояния

NFS4 – сервер с состоянием (т.е. тут хранится информация о клиенте) (оптимизирует в случае наличия большого количества клиентов)

В NFS3 есть команда lookup, возвращающая дескриптор

В NFS4 есть команда open – которая возвращает дескриптор, и открывает файл, и запоминает немного о пользователе, а в конце нужно сделать close

Иногда могут возвращаться несколько дескрипторов (например, если у нас есть несколько копий файла), то тогда можно будет читать поочерёдно у этих серверов.

Способов доступа для нескольких пользователей к одному файлу – несколько:

1. Механизм транзакций – каждое обращение к файлу оформляется в виде транзакций, и всё, что внутри транзакции – будет как в базе данных – подвергнуто ACID (неделимость, выполнимость полностью или никак, …)

Очень часто мы читаем весь файл, рандомный доступ бывает редко, а поэтому можно пытаться выполнять на сервере упреждающее чтение и грузить файл с жёсткого диска в память.

При выполнении операции open в NFS4

1. Сообщения могут быть короче и не содержать в себе ту кучу системных данных, делающих в NFS3 запрос самодостаточным
2. Проще убедиться в достоверности запроса
3. Если сервер перегружен, и не успевает отвечать на клиентов, то тогда он может собрать запросы от одного клиента в кучку и разбираться с ними по частям, что хорошо

Если клиент ждал и уже успел послать повторный запрос, то сервер может уже знать, что этому клиенту он уже отправил данные, и ему больше не нужно

А если запрос потерялся и от клиента пришёл повторный вопрос, то наш ответ можно попытаться взять из кеша

1. Возможна операция захвата
2. Эффективность выше (В NFS4 некоторые запросы можно обрабатывать локально, но для этого нужно помнить, кому и какие права были даны)

Существуют различные блокировки на время (хахват файла):

Lock

Lockt

Renew

Если пользователь по истечении времени не сделал renew, то блокировка будет снята

Предположим, что сервер упал, и потерял текущие блокировки, поэтому серверу нужно как-то восстановить состояние, поэтому есть некоторое время амнистии, когда сервер принимает только команды renew (период должен быть дольше, чем квант времени для монопольной работы с файлом)

Благодаря этим renew – сервер узнаёт, кто сейчас пишет в файл

У NFS3 – нету ограничения памяти для таблиц, и можно запустить любое количество пользователей, нету open и close, устойчив к ошибкам

**Лекция 8**

Рассмотрим механизм кеширования и размножения файлов

Места хранения файла:

Оперативная память клиента и сервера, и жёсткие диски клиента и сервера

1. Нужно отправить запрос от клиента к серверу
2. Сервер должен после этого обработать файл, он разобьёт его на части и потом сделает самую дорогую операцию – переместит головку (скорость самого чтения – сотни гигов в секунду)
3. …

Кешировать можно:

1. Дескриптор файла

Есть следующие способы синхронизации, если файл в нескольких местах:

1. При изменении файла в одном месте сразу делать широковещательное оповещение
2. Если кто-то что-то делает, то сначала он должен спросить сервер, не было ли изменений

Много прослушал

Способы размножения файлов:

1. Ленивое
2. Симметричное

Появляются проблемы: есть много копий данных,

Метод размножения главной копии – выделяется главный сервер, на нём ведётся журнал всех изменений, которые были произведены.

Короче говоря, я всё прослушал

**Лекция 9 Hadoop**

Почему появились базы данных:

1. Сейчас стало информации совсем много, и она совсем не структурирована
2. Нужно как-то распараллеливать считывание и запись информации, для увеличения скорости работы с большим количеством информации

В hadoop пользователь фактически работает со списками

Hadoop использует технологии MapReduce

Отличительная особенность в том, что при использовании MapReduce – полезно лишь для очень больших размеров данных, по сравнению с обычными реляционными базами (например петтабайты)

MapReduce лучше работает с однократной записью, но многократным чтением

Целостность у MapReduce низкая

Масштабируемость MapReduce – линейная, т.е. мы можем увеличивать количество серверов и линейно с этим будет увеличиваться производительность (и линейно падать вместе с ростом количества информации) (крайности соотношений количества данных и количества серверов – не надо рассматривать)

MapReduce в первую очередь был создан google

Hadoop – является opensource

Недостатки работы в традиционной базе данных – тяжело таскать данные на каждый вычислитель, так что было бы хорошо уметь таскать процессы и саму обработку к данным.

Задачи Hadoop:

1. Обработка большых данных
2. Автоматическое распределение выполняемых функций между железом
3. Автоматическая обработка отказов и распределение по оборудованию

Hadoop имеет 3 части:

1. Common – для стыковки с разными файловыми системами
2. HDFS – сама файловая система
3. MapReduce – на java

Архитектура hadoop HDFS

Есть сервер namenode, который содержит в себе информацию и содержит в себе все изменения, которые время от времени сбрасываются на диск (которых несколько копий, для страховки информации (и в то же время читать можно с нескольких сразу)), ещё ведётся журнал, чтобы когда namenode упал, то всё было не очень плохо, ибо из журнала что-нибудь восстановится

Namenode пытается всё хранить в памяти.

DataNodes – это те, которые хранят всю информацию на жёстких дисках, (разные куски информации), и сами datanodes объединяются в Rack, и иноформация имеется в нескольких копиях на разных реках, но как именно она распределена в реках – не регламентируется.

MapReduce работает со списками, и по факту там как в lisp (оттуда и украли) – лишь 2 разных функции – map и reduce. Map отображает данные из одного списка в другой (немного обрабатывая, т.е. выбирая нужные данные, из может быть сложной структуры данных, которая хранится в качестве элемента списка), Reduce выбирает элемент их списка (используя некоторую функцию, которая анализирует входной список, и выдаёт ответ).

Такую работу hadoop может запускать параллельно на разных кусках данных, а потом их можно объединять, сначала их упорядочивают по ключу, а потом снова обрабатывают как список.

Работа hadoop при чтении получается такой:

1. Параллельно для разных данных делаем map – (тут берутся лишь нужные данные)
2. Shuffle process – которые пересобирает списки для дальнейшей обработки, которые выдал map – это может быть например сортировка по ключу
3. Reduce – выбирается нужный элемент уже из сформированных списков

Промежуточная информация (промежуточные списки) – не сохраняются.

JobTracker – он разбивает задачи на подзадачи, например он может сам выбирать, на какие блоки будут биться данные при параллельно обработке, он разбирается, можно ли разбить нашу задачу при выполнении map на подзадачи (хотя в основном он преимущественно распределяет нагрузку, пытаясь манипулировать разбиением данных)

JobTracker управляет TaskTracker-ами. В общем его основная задача – распределение нагрузки.

Из данных перемещаются лишь входные списки для Reduce, а их объём мал.

Hadoop Plugin для Eclipse – для работы на java

Есть Hadoop Streaming – позволяющий программировать на любых других языках: shell, python, ruby, …

Системы на основе hadoop – pig, hbase, Cassandra, Hive (попытка сделать интерфейс hadoop как в sql), Mahout, …

**HDFS Система** – ёмкость, производительность, надёжность.

Существуют специальные серверы, которые делают для хранения большого количества информации, и для наличия огромного диска, но это очень дорого стоит.

Google File System – суть в том, что мы пользуем обычные ПК, но при правильной организации, получаем большую скорость доступа за счёт распределённости.

Получается, что есть все требования – стоимость, надёжность, скорость, объём.

Ограничения HDFS:

1. Случайный доступ работает медленно, быстро работает считывание большого количества блоков
2. Модель использования – один раз записали, много раз считали
3. Уровень виртуальной файловой системы – отсутствует, т.к. нас тут уже не сильно заботит прозрачность сети для клиентского пользования.

Поэтому тут нету POSIX, и следовательно не работает монтирование файла, ls, cp, mkdir, …

1. Кеширование не используется, т.к. объёмы памяти огромны, а потому негде столько закешировать, + у нас пакетная обработка, а потому запросы, а значит и данные не особо повторяются

Namemode – управляющий узел, он занимается стыковкой, разрешением пространства имён, …

Datanode – просто хранит данные

Обычно репликация в мире равна 3-м – т.е. один и тот же блок хранится в 3-х местах.

NameNode – знает, на каком узле хранения экземпляры каких данных имеются

Rack Awareness – данные обычно пытаются распределить в нескольких разных шкафах, чтобы если шкаф умер, то другой был жив.

Для работы с hdfs используются специальные команды, например /bin/hadoop dfa -ls

К hdfs можно подключиться из терминала или из интернета.

Все пути либо полные, либо из домашней директории пользователя

Как обычно система разграничения доступа – дискреционная.

r/w – для файлов - стандартно

x для файлов не используется

r- для каталога это чтение

w – для каталога это создание файлов, папок

x – возможность заглядывать внутрь рекурсивно

нету setuid, setgid, sticky bit

Существует api для java, чтобы работать с hdfs

**Лекция 10**

Лекция по обеспечению надёжности в распределённых системах

Читай по слайдам лектора (7-й номер)

**Лекция 11**

**Обзор технологии параллельного программирования MPI**

Была хорошая презентация

Тег – есть для каждой операции точка-точка (используется для определения того, что пришло к нам (фактически это нумерация))

**Лекция 12**

Автоматизация разработки параллельных программ.

Вроде бы эта лекция всегда читалась, хотя в курс не входила, но выучить всё равно стоит.

Смотри слайды.