**Теоретический минимум по Распределённым Операционным Системам**

The History (Of magic) (старые 50-е, добрые 80-е и лихие 90-е)

Достоинства распределённых систем – дешевле наращивать, нет потолка в наращивании, поддержка естественной распределённости по территории, надёжность, прозрачность (расположения, миграции, размножения, параллелизма, конкуренции), гибкость, масштабируемость (децентрализация)

Недостатки – жуть как сложно не напороться на падение, deadlock, …

Виды операционных систем –

1. Сетевые ОС – это скорее автономные системы, объединённые сетью, можно ввести задание в чью-нибудь очередь, заглянуть в чужую файловую систему.
2. Распределённые ОС – общий коммуникационный механизм для процессоров, одинаковое видение файловой системы, виртуальный мультипроцессор, … - как будто один комп.
3. Мультипроцессорная ОС – и правда один комп) – единая очередь выполняющихся процессов, одна файловая система.

Транспьютер – штука с 4-мя вх/вых

Процессы, нити.

Процессы могут общаться через общую память, и через сообщения.

2 вида синхронизации – взаимное исключение критических интервалов (частный случай – КС (критические секции)) (семафоры, например), и координация процессов (сообщения, например).

Взаимное исключение критических интервалов

1. Требования –
   1. Никто не ждёт вечно
   2. Если КС свободна – то всегда можно зайти
   3. Нельзя зайти в КС одновременно нескольким
   4. Никаких предположений о скоростях процессоров
2. Механизмы –
   1. Алгоритм Деккера (исключительно на 2 процесса) (активное ожидание - плохо)

|  |  |  |
| --- | --- | --- |
| int turn;  boolean flag[2];  proc (int i)  {  while (TRUE)  {  <вычисления>;  enter\_region (i);  <критический интервал>;  leave\_region (i);  }  } | void enter\_region (int i)  {  try: flag[ i ]=TRUE;  while(flag[(i+1)%2])  {  if (turn == i) continue;  flag [i] = FALSE;  while (turn != i);  goto try;  }  } | void leave\_region(int i)  {  turn = (i+1) % 2;  flag [i] = FALSE;  } |

* 1. Алгоритм Петерсона (исключительно на 2 процесса) (активное ожидание - плохо)

|  |  |
| --- | --- |
| void enter\_region( int i )  {  int other;/\*номер другого процесса\*/  other = 1 - i;  flag [i] = TRUE;  turn = i;  while(turn==i && flag[other] == TRUE)  /\* пустой оператор \*/;  } | void leave\_region (int i)  {  flag [i] = FALSE;  } |

* 1. TEST\_and\_SET\_LOCK – (активное ожидание - плохо)

|  |  |
| --- | --- |
| enter\_region:  tsl reg, flag  cmp reg, #0 /\*сравниваем с нулем\*/  jnz enter\_region /\* если не нуль - повторяем попытку \*/  ret | leave\_region:  mov flag, #0 /\* присваиваем нуль\*/  ret |

* 1. Семафоры Дейкстры
     1. Функция запроса семафора P(s):

[ if (s == 0) <заблокировать текущий процесс>; else s = s-1; ]

* + 1. Функция освобождения семафора V(s):

[ if (s == 0) <разблокировать один из заблокированных процессов>; else s = s+1; ]

Задача обедающих философов, задача читателей и писателей.

* 1. События (post, wait, clear)

1. Планирование процессов –
   1. Поменьше нужно переключать контекст, перемещать страницы виртуальной памяти, портить кэш.
   2. Не нужно переключаться на другой процесс, когда текущий выполнял критическую секцию, и его многие ждут.

Для улучшения характеристик, делают:

1. Совместное планирование – чтобы все процессы одного приложения одновременно выбирались и снимались с процессоров.
2. Находящиеся в критической секции процессоры – почти не прерываются.
3. Задачи не перемещаются линий раз между процессорами.
4. Учёт «советов» программы (например, есть специальные директивы в ОС Mach)

Коммуникации в распределённых системах –

1. Для ускорения обмена информацией в транспьютерной решётке можно использовать – конвейер и параллельные маршруты.
2. MPI – буферизуемые (не забудьте выделить буфер своими лапками), по готовности, синхронные, блокирующие (mpirun –mp3 prog)
   1. Коммуникация точка-точка (послать, получить, подождать, проверить успешность, отменить (all, some, any) (test, wait, probe) (send, receive))
   2. Коллективные операции (broadcast, scatter, gather, allgather, alltoall, reduce (yj функции только предопределённые – max, min, sum, xor, …))
   3. Группы процессов
   4. Коммуникационные контексты
   5. Топология процессов
   6. Простой способ создания процессов для модели SPMD

В MPI нету поддержки нитей, работы с разделяемой памятью и всяких таймеров.

Коммуникатор, группа, контекст (область видимости).

1. MPI-2
   1. Динамическое создание и удаление процессов
   2. Средства синхронизации для работы с общей памятью
   3. Параллельные операции ввода/вывода
2. Parallel Virtual Machine –
   1. Надстройка над ОС
   2. Оъединяет несколько рабочих станций
   3. Задачи пользователя динамически создаются на указанных процессорах.

Просто + динамическое добавление процесса к группе выполняемых процессов.

ООчень медленно + функциональная ограниченность.

1. Синхронизация времени (все процессы посылают всем своё время, и у всех процессов выбирается максимальное)
2. Выбор координатора
   1. «Задира» - шлём всем с номером старше «выборы», если кто-то ответил – то дальше выборы проводит он, если никто не ответил – то выиграл и шлём всем «координатор»
   2. Круговой алгоритм – все по кругу говорят «выборы» и указывают всех, кто был передан им в списке, когда им сказали и добавляют туда себя, не отвечающие – пропускаются, если кто-то видит, что уже есть в списке, то теперь рассылает слово «координатор», которое проходит по кругу.
3. Взаимное исключение
   1. Централизованный алгоритм (для входа в критическую секцию – нужно стать в очередь у координатора)
   2. Децентрализованный алгоритм на основе временных меток (всем шлём своё время и спрашиваем разрешение у кого-нибудь – со временем процессы – ответят. Все остальные процессы шлют «ОК» - если они в критическую секцию входить не хотят, или их время больше чем присланное, если только вышли из секции – то сказать всем желающим «ОК»)
   3. Алгоритм с круговым маркером (маркер просо передают по кругу – у кого маркер, тот и входит в секцию, если хочет)
   4. Алгоритм широковещательный маркерный (Suzuki-Kasami) (Маркер содержит очередь запросов, и массив с номерами последних выполненных запросов (в массиве – по элементу на каждый процесс). У каждого процесса есть массив, где он помнит – какой последний номер запроса попросил тот или иной процесс. Желающий процесс – шлёт всем номер своего желаемого запроса входа в критическую секцию. Тот, кто имеет свободный маркер сверяет – как раз ли попросили у него только что следующий номер запроса (сверив что он на 1 больше, чем записано в массиве маркера). А если ты только вышел из секции, то обрабатываешь все просьбы, добавляешь в очередь и после этого отдаёшь маркер)
   5. Алгоритм древовидный маркерный (Raymond) (Каждый процесс помнит, в каком направлении маркер, и каждый процесс при желании его заполучить шлёт в ту сторону «запрос», получив подобное, каждый запомнит у себя в очереди, в какую сторону нужно было отдать маркер и зарепостить туда запрос. Каждый процесс при получении маркера выполняет КС, а потом отдаёт в нужном направлении, вычеркнув направление из совей очереди, если в очереди что-то ещё осталось, то снова репостим в нужном направлении запрос)
   6. Измерение производительности
      1. MS\CS – количество операций приёма сообщений, для прохождения КС.
      2. TR – время от появления запроса до получения разрешения на вход в КС.
      3. SD – синхронизационная задержка (время между выходом из КС одного и входа другого)
      4. LL – в условиях низкой загрузки, LH – в условиях высокой загрузки

Распределённые файловые системы –

1. Главные цели – сетевая прозрачность, высокая доступность.
2. Файловый сервис/сервер
3. Модели доступа – «загрузка/ разгрузка» и «удалённый доступ»
4. Интерфейс сервера директорий
   1. Всё что касается управления директориями, расположением файлов в них, удаление, перемещение, …
   2. Прозрачность именования (прозрачность расположения и прозрачность миграции) (машина + путь, или монтирование или единое пространство имён)
5. Двухуровневое именование – символьное имя, и системное имя.
6. Семантика разделения файлов –
   1. Неизменяемые файлы
   2. Семантика сессий
   3. Транзакции (как бд)
   4. UNIX-семантика (т.е. когда читаем – видим последнее изменение)
7. Типичное использование файлов
   1. Большинство файлов – малого размера
   2. Чтения гораздо больше записи
   3. Чтение и запись обычно последовательны
   4. Не многие файлы действительно разделяются
8. Клиент и сервер – могут быть разных типов, а могут быть абсолютно идентичны.
9. Сервера с состоянием – короче сообщения, быстрее, упреждающее чтение, возможен захват файла, проверка достоверности. Период амнистии (после падения мы принимаем лишь renew, но не open)

Без состояния - устойчивость к ошибкам, нет открыть/закрыть, не нужно помнить служебную таблицу для клиентов, нет проблем с падением клиента или сервера, нет ограничения на число открытых файлов.

1. Кэширование
   1. На дисках сервера – всё равно долгий доступ
   2. В памяти сервера – кеш не такой уж и большой – как выталкивать, сколько читать, …
   3. В диске клиента – сильно сложнее (когерентность), а может не дать выгоды.
   4. В памяти клиента – (тоже не просто, но всё же)
      1. Кеширование в ядре
      2. Кеширование в каждом процессе
      3. Кеш-менеджер в виде отдельного процесса
2. Консистентность кеша
   1. Сквозная запись
   2. Отложенная запись (по времени)
   3. Сброс кеша при закрытии файла
   4. Через централизованное управление (выдерживание семантики UNIX – т.е. при чтении почитывается последнее записанное) (не эффективно, не надёжно, плохо масштабируется)
3. Размножение –
   1. Надёжность
   2. Доступность
   3. Распределение нагрузки
4. Работа с размножением –
   1. Явное размножение – клиенту выдают несколько дескрипторов.
   2. Ленивое размножение – копии создаются пока сервер не занят, и так же поддерживает их корректность – когда не занят.
   3. Симметричное размножение – все операции вызываются и выполняются на всех серверах одновременно.
5. Протоколы коррекции
   1. Метод размножения главной копии (есть главный сервак – ему и посылаем всё, а он уже сам корректирует остальные копии)
   2. Метод одновременной коррекции – все изменения шлются всем.
   3. Метод голосования – для успешного чтения и записи – нужно преодолеть некоторый порог
6. NFS – можно как монтировать, так и читать отдельные файлы, использует ассиметричное шифрования для аутентификации пользователя, передача информации – блоками по 8К
   1. NIS – сетевой информационный сервис – используется для контроля доступа, выдаёт значения кодов, отображает имена в сетевые адреса, …
   2. NFS3 – сервер без состояния (lookup)
   3. NFS4 – сервер с состоянием (open/close)

Преимущества DSM –

1. Программисту удобно (потому что не нужно мучиться с MPI и потому что удобно передавать структуры, которые внутри себя могут иметь ссылки на другие структуры, …)
2. Линейное увеличение количества памяти вместе с увеличением количества узлов, + потолка сверху нету
3. Достаточно переносимо

Способы реализации DSM –

1. Централизованный сервер – у него все данные, и именно к нему все обращаются каждый раз.
2. Миграционный алгоритм – странички перемещаются туда сюда в монопольном режиме
3. Размножение для чтения – если кто-то запишет что-то в свою копию, то остальным будут посланы сообщения о том, что их копии невалидны, и если они хотят читать, то они снова попросят нужную страничку.
4. Размножение для чтения и записи – через центральный сервак или широковещание – рассылаются все изменения.

Модели консистентности DSM

1. Строгая консистентность (события все видят так, как они и происходили)
2. Последовательная консистентность (все видят события в некоторой одной и той же последовательности) (можно делать как через широковещание, так и через централизованный сервак)
3. Причинная консистентность (отсылая всем новые значения, мы так же отсылаем и те, от которых мы зависим, чтобы остальные сначала дождались их, а уже потом применили наши изменения)
4. PRAM консистентность (каждый локально нумерует свои операции записи и рассылает и всем, после чего остальные могут всегда восстановить правильный порядок для данного процессора)
5. Процессорная консистентность (тут дополнительно к PRAM, нумеруются и последовательности обращений процессоров к конкретной разделяемой переменной (нужно пользовать либо широковещание, либо централизацию))
6. Слабая консистентность (Есть синхронизационные переменные – за ними следят по модели последовательной консистентности, а остальные расшаренные переменные – их «итоговые изменения» (для каждого процессора между каждыми синхронизационными) – будут синхронизироваться лишь вместе с самими синхронизационными)
7. Консистентность по выходу (Есть вход и выход в критическую секцию – соответствующие команды для синхронизационных переменных. И синхронизация происходит:)
   1. Энергичная – сразу по выходу из секции – всем кому надо рассылаются изменения
   2. Ленивая, каждый кто собрался входить в критическую секцию – идёт и просит свежую версию от последнего использовавшего.
8. Консистентность по входу
   1. Отличие первое – тут жёстко программным образом программист регламентирует какие расшаренные переменные какими синхронизационными переменными отслеживаются.
   2. Можно делать отдельно не монопольный доступ для тех, кто желает только читать и монопольный для тех, кто желает писать.

Конструкторские решения DSM –

1. Страничная DSM (если размер страницы совпадает с размером страниц в локальной памяти – то удобно встраиваться в механизм)
2. DSM на базе разделяемых переменных (просто вручную указано, какие переменные - разделяемые)
3. DSM на базе объектов (т.е. работаем через специальные методы специального класса)

Надёжность систем –

1. Отказы бывают – случайными, периодическими и постоянными.
2. Проблемы восстановления – сообщения сироты, эффект домино, потеря сообщений, проблема бесконечного восстановления.
3. Контрольные точки бывают – консистентные (для любого приёма была фиксирована и посылка) и строго консистентные (нет никаких передач и приёма).
4. Способы создания контрольных точек –
   1. Синхронная фиксация (есть тот, кто инициирует постановку пробной точки, а потом утверждает её для всех)
   2. Асинхронная фиксация (откатываемся, пока не случится успех)
5. Отказоустойчивость
   1. Горячий резерв
   2. Протокол голосования (количество успешных записей должно быть больше порога, аналогично и для чтения)
   3. Протокол принятия коллективного решения
      1. Протокол принятия единого решения – история про 3 армии, численностью в 5, 3 и 3. (ненадёжный канал связи)
      2. Протокол принятия согласованного решения – история про генералов, 1/3 среди которых – предатели. (ненадёжные процессоры)

Надёжная неделимая широковещательная рассылка сообщений. (разослали всем своё время, те разослали своё, потом выбрали максимум и разослали всем, если по по пути сбой, то кто-то должен всё начать заново)

Hadoop и MapReduce (GFS, …) – состав – Common (компоненты, стыковка с файловой системой, …) + HDFS + MapReduce (под java, …)

1. Концепции –
   1. Функциональное программирование (подобно lisp) + перемещение вычислений к данным
   2. Данные почти не пишутся – очень много читаются
   3. Простота использования при автоматическом распараллеливании
   4. Перемещаются лишь промежуточные списки – их объём мал.
   5. Дёшево и сердито, всёгда что-то ломается
2. HDFS – Namenode – знает где и что лежит, DataNode – в нём лежит, Rack1 – шкафы, в которые собраны DataNode

Адресное пространство общее, файл разбивается на независимые блоки, которые реплицируются

Права доступа – дискреционные, команды для работы с файловой системой – свои.

1. MapReduce – Последовательно делается Map данных с получением промежуточных списков, потом их Shuffle (перегруппировка), а потом их Reduce – и результат reduce в совокупности – и есть ответ.
2. Архитектура – Клиент даёт задачу JobTracker, тот её может соптимизировать (сам решить, как данные разбить, …) и отдаёт TackTracker – который выполнит нужную операцию.

Примеры систем hadoop – Hive, Pig, Cassandra