Сериализуемое многопоточное исполнение очереди

Баукин Антон

- ☑ асинхронные запросы
- ☑ сериализуемое исполнение
- ☑ хэш-инвариант данных
- ☑ блокировки, их эскалация и тупики
- ☑ изоляция образов
- ☑ синхронизация без ожидания

Ссылочная задача “Проблема должников”

Данные: клиенты со счетами и списками долгов

Запросы к системе:

- ☑ покупка, если есть вся сумма
- ☑ кредит на покупку, если не хватает

Особенности задачи:

При получении денег система производит возврат долгов кредиторам. Этот процесс имеет рекурсивный характер, и поэтому полный перечень клиентов, к которым обратится система не ограничивается только двумя из запроса, и при входе в запрос заранее не известен.

Сбой при выполнении одного лишь запроса способен изменить всё будущее системы и результирующие остатки на счетах всех клиентов, а не только тех двух из запроса.

Генератор тестовых запросов

При запуске генератор вводит начальные суммы на счетах и создаёт последовательность запросов к системе, выбирая случайные пары клиентов, тип запроса и его параметры. Очередной запрос не зависит от предыдущих.

Решение с блокировками

В задаче всегда блокируется как минимум два клиента. Недетерминированная последовательность блокирования неизбежно приведёт к тупикам. Классическое решение Дейкстры состоит в том, чтобы блокировать по порядку ресурсов, здесь: по номеру клиента.

Феномены преждевременного выполнения покупок:

- ☑ если из двух запросов списания средств у одного клиента выполнить сначала второй, и средств не хватит на списание в первом, то итоговый инвариант нарушается.
- ☑ если при недостатке средств попытаться выполнить списание до зачисления, после которого средств уже хватит.

Блокировать всё и сразу

Кажется правильным, что достаточно блокировать пару клиентов из запроса и клиентов из списка кредиторов клиента-получателя. К сожалению, внутри рекурсивного вызова, мы уже не можем блокировать списки кредиторов, поскольку нарушим порядок блокировок внешнего вызова, что приводит к тупику.

Мы не можем построить замыкание списка блокировок, чтобы во внешнем вызове заблокировать сразу всех, так как этот рекурсивный процесс сам требует блокировать клиентов из списков долгов.

Решение без блокировок

Все синхронизационные примитивы имеют в своей основе некотрый атомарный механизм, реализуемый аппаратно, такой как проверка-и-замена (CAS, compare-and-swap) регистра. Неизбежно возник вопрос, можно ли придумать такой подход к структурам данных, чтобы запросы к ним выполнялись без блокировок (lock-free) или даже без задержек (wait-free), применяя атомарные операции.

Основная особенность блокировок в том, что ожидающие потоки могут быть переведены в сон, а разбужены через тысячи тактов процессора (сотни наносекунд), что приводит к большим случайным задержкам в обработке одиночных запросов. Прежде, для однопроцессорных мультизадачных систем это было единственным спасением от монопольного захвата ресурсов одной задачей. Поскольку к настоящему времени вычислительные мощности выросли в сотни раз, во многих случаях для потоков более эффективно дождаться входа в небольшую критическую секцию в активном ожидании (через спин-блокировку).

Отметим, что реализация алгоритмов без ожидания или без блокировок требуют особой тщательности в вопросах управления памятью, а также знания деталей целевой аппаратной платформы: иначе их производительность будет даже меньше решений на блокировках.

Упреждающее исполнение очереди

Упреждающее исполнение команд (суперскалярная архитектура) реализовано в массовых процессорах Intel Pentium, Sun SuperSPARC, Motorola MC88110 с начала 90-х годов, хотя сама идея возникла и была реализована в больших ЭВМ десятки лет до этого. Применим её к нашей задаче.

Пусть есть только два потока, и условимся, что второй всегда исполняет из очереди запросы, следующие за тем, который исполняет первый. Если запрос второго независим по данным от запроса первого (и всех запросов между ними в савокупности), то допустимо самостоятельное исполнение обоих запросов.

Но что значит на практике: проверить, что нет зависимости? Это заглянуть в структуры данных первого потока, что само требует синхронизации. Или первый поток сам должен выставить в общий пул сведения, какие у него данные.

Версионность разделяемых данных

Рабочий поток упреждающего исполнения запросов оперирует не только данными, локальными для запроса, но и разделяемыми данными. Чтобы не нарушать их целостность, поток создаёт их изолированную копию. Стандартной практикой здесь является копирование данных при первой попытке их изменения. Доступ контролируется через пару блокировок на чтение-запись.

Для задачи о долгах разделяемыми ресурсами является для каждого клиента список долгов и счёт. Наиболее эффективным, надёжным и простым решением будет копировать весь объект данных клиента при первой необходимости в них.

Хэширование данных предварительного исполнения

Краеугольным камнем рассматриваемого решения является создание хэшей исходных данных, полученных рабочим потоком, и вычисленных до их изменения. Поток сохраняет результаты своей работы вместе с хешами, но не изменяет глобальные данные, если его задача не является первой из ещё не выполненных задач (мастер-задачей).

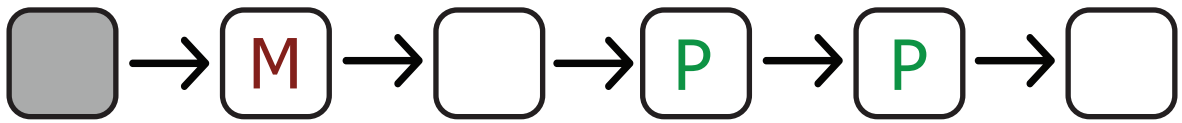
При входе в мастер-задачу поток прежде всего проверяет, не выполнена ли она. Если да, то он сверяет хэши данных, которые были исходными для упреждающего исполнения. Предполагается, что результаты исполнения при тех же исходных данных не будут отличаться. Если хэши совпали, то поток применяет готовые результаты к глобальным данным.

Такая схема выполнения очереди эффективна, если вычисление хэшей и копирование результатов выполняется быстрее, чем полное исполнение задачи.

Мастер-поток исполнения очереди

Имеет смысл наделить один из рабочих потоков особыми полномочиями и поведением. Этот поток движется по очереди всегда последним, не оставляя позади ни одного невыполненного запроса. Также, он владеет и напрямую изменяет оригинальную копию разделяемых данных. Назовём такой поток мастером исполнения.

Тогда многопоточное исполнение очереди преобретает следующий вид. Каждый из рабочих потоков выбирает очередную доступную задачу, следующую за задачей мастера. Перед входом в неё, поток получает образ данных мастера. Если мастер догоняет рабочий поток, то его результаты последнего не учитываются.



Рабочие потоки обгоняют мастер лишь на несколько запросов. Чем дальше задача от мастер-задачи, тем более вероятно неверное её исполнение, и поэтому нет смысла забегать далеко вперёд. Достигнув предела, вторичные потоки ожидают, когда мастер сдвинется вперёд. Когда результаты предварительного исполнения верны, мастер быстро догоняет рабочие потоки, поскольку просто сверяет хэши и применяет результаты.

Хэширование данных клиента

```
411
412     /* Synchronization */
413
414     /**
415      * Compares the data of the same client c
416      */
417     public boolean equals(Hash hash) { return
421
422     public Hash    hash(Hash hash)
423     {
424         hash.update(id);
425         hash.update(money);
426         debts.hash(hash);
427
428         return hash;
429     }
430
431     /**
432      * Returns the Hash of this Client
433      * re-calculating if Client is updated.
434      */
435     public Hash    hash()
436     {
437         if(!updated)
438             return this.hash;
439
440         this.hash.reset();
441         this.hash(this.hash);
442         updated = false;
443
444         return this.hash;
445     }
```

Доступ к очереди для мастер-потока

```
847     /* Queue Access */
848
849     public Data master(Data data)
850     {
851         //?: {release previous request}
852         if(data != null)
853             datas[data.index] = null; //<-- reduce GC pulsation
854
855         final int x = cursor[0];
856         if(x >= requests.length)
857         {
858             data.locked.set(2);
859             return null;
860         }
861
862         //~: advance the cursor
863         cursor[0]++;
864
865         //HINT: Support thread ID starts with 1. Having 1 Support
866         // next pre-execute position is +2 after this (x) request.
867         // +2 (not +1) is better as when Support will take it,
868         // Master will be about to take +1.
869
870         //~: move prepare position
871         cursor[1] = x + THREADS-1;
872
873         //~: unlock waiting Support threads
874         if(data != null)
875             data.locked.lazySet(2);
876
877         return datas[x];
878     }
```

Доступ к очереди для рабочего потока

```
880     public Data support(final int id)
881     {
882         int offset = id;
883         int presee = 0;
884
885         //c: queue competition cycle
886         while(true)
887         {
888             //~: position to pre-execute (starts with 1)
889             final int index = cursor[1] + offset + presee;
890
891             //?: {the queue is almost done}
892             if(index >= requests.length)
893                 return null; //<-- thread exit
894
895             //?: {master is ahead}
896             final Data data = datas[index];
897             if(data == null)
898             {
899                 offset += THREADS-1;
900                 continue;
901             }
902
903             //?: {this item is not locked} take it
904             if(data.locked.compareAndSet(0, 1))
905                 return data;
906
907             //?: {pre-see limit not gained}
908             if(presee++ <= PRESEE)
909             {
910                 offset += THREADS-1;
911                 continue;
912             }
913
914             //~: spin on this item
915             while(data.locked.get() != 2);
916         }
917     }
```

Исполнение задач рабочим потоком

```
1243    protected Client client(int id)
1244    {
1245        //?: {found it in the local cache}
1246        Client c = cache.get(id);
1247        if(c != null) return c;
1248
1249        //~: load from the database & cache it
1250        c = queue.database.copy(id);
1251        cache.put(c.id, c);
1252
1253        //~: remember the initial version
1254        initial.add(c.id);
1255        initial.add(new Hash(c.hash()));
1256
1257        //~: add to the results
1258        results.add(c);
1259
1260        return c;
1261    }
1262
1263    private void    execute()
1264    {
1265        //~: clear the cache
1266        cache.clear();
1267
1268        //~: allocate execution data
1269        initial = new ArrayList<>(4);
1270        results = new ArrayList<>(2);
1271
1272        //~: execute the request
1273        execute(data.request);
1274
1275        //~: calculate hashes
1276        for(Client c : results)
1277            c.hash();
1278
1279        //~: save data results
1280        data.initial = initial.toArray(new Object[initial.s
1281        data.results = results.toArray(new Client[results.s
1282    }
```

Доступ к очереди для мастер-потока

```
1141  protected Client client(int id)
1142  {
1143      //?: {found it in the local cache}
1144      Client c = cache.get(id);
1145      if(c != null) return c;
1146
1147      //~: load from the database & cache it
1148      c = queue.database.copy(id);
1149      cache.put(c.id, c);
1150
1151      return c;
1152  }
1153
1154  private void    execute()
1155  {
1156      cache.clear();
1157
1158      //?: {the resulting data may be applied}
1159      if(consistent())
1160          queue.database.assign(data.results);
1161      //~: execute again
1162      else
1163      {
1164          //~: execute the request
1165          execute(data.request);
1166
1167          //~: assign all the changes to the database
1168          queue.database.assign(cache.values());
1169      }
1170  }
```

```
1172  private boolean consistent()
1173  {
1174      if(data.results == null)
1175          return false;
1176
1177      //?: {global copy differs}
1178      if(!queue.database.same(data.initial))
1179      {
1180          miss++; //<-- support work is wasted
1181          return false;
1182      }
1183
1184      hits++; //<-- support work is not wasted
1185      return true;
1186  }
```

Результаты

Только мастер
```
      45     0.0   0.0  397592016B65C1CF9216A1F0BB443B7F
```

Мастер + 1
```
      36   43.6   1.0  397592016B65C1CF9216A1F0BB443B7F
```

Мастер + 2
```
      31   38.2   3.5  397592016B65C1CF9216A1F0BB443B7F
```

Только мастер с задержкой
```
     271    0.0   0.0  397592016B65C1CF9216A1F0BB443B7F
```

Мастер + 1 с задержкой
```
     205   63.3   0.7  397592016B65C1CF9216A1F0BB443B7F
```

Мастер + 2 с задержкой
```
     172   60.7   0.8  397592016B65C1CF9216A1F0BB443B7F
```

Условия: CPU Intel i7 950 (4); 100 клиентов, 200 тысяч запросов.
Задержка при первом обращении к данным клиента составила 2 мс,
что сопоставимо со временем запроса к локальной СУБД.