Р. И. Идрисов

ПАРАЛЛЕЛИЗМ В JAVASCRIPT[1]

ВВЕДЕНИЕ

О преимуществах параллельного исполнения на сегодняшний день уже можно подробно не рассказывать, поскольку эта тема стала достаточно очевидной. Тенденции таковы, что даже мобильные телефоны оснащаются многоядерными процессорами. Согласно ресурсу Steam[2], который собирает статистику по конфигурации компьютеров собственных клиентов, пропорция однопроцессорных, двухпроцессорных и четырёхпроцессорных персональных ПК изменилась с 14.1%, 56.9% и 27.24% (август 2010) до 5.1%, 48.21% и 43.54% (январь 2012). Есть и другая тенденция, заключающаяся в том, что большое количество приложений переходят в браузеры (программы просмотра html-документов); это обеспечивает некоторую универсальность вместе с неизбежными ограничениями.

В связи с этим возникает вопрос о возможности эффективного использования современных персональных компьютеров при помощи браузера и JavaScript в частности. Существует несколько способов измерить производительность JavaScript, можно воспользоваться тестом от Mozilla[3], который называется v8. Этот тест включает в себя много частей, которые покрывают потенциальные способы использования JavaScript для большого количества вычислений, но если попытаться проверить степень параллелизма этих вычислений, выяснится, что все они исполняются только на одном процессоре/ядре. Тем не менее JavaScript содержит модель конкурентного исполнения. Попробуем выполнить следующее многопоточное приложение:

```
function startThread(num) {
      var l=0, dateObj=new Date(),
          startTime=dateObj.getTime();
      if (num < 1) return;
      console.log("Thread " + num + " started");
      setTimeout('startThread('+(num-1)+');', 100);
      while (dateObj.getTime()-startTime<3000) {
              dateObj=new Date();
              l+=Math.random();
      }
      console.log("Thread " + num + " ended (l=" + l + ")");
}
startThread(10);
```

.

[1] Работа поддержана грантом РФФИ № 12-07-31060 мол_а.

[2] http://store.steampowered.com/hwsurvey

[3] http://dromaeo.com/?dromaeo|sunspider|v8

Это приложение создаёт 10 вычислительных потоков с интервалом в 0.1 секунды, каждый из которых работает в течение трёх секунд. Результаты удивляют, поскольку потоки выполняются полностью последовательно для всех популярных на данный момент браузеров[4]. В случае с IE потребуется заменить console.log на document.write или alert, но исполнение остаётся последовательным.

Таким образом, несмотря на теоретическую возможность, текущие реализации языка не поддерживают исполнение нескольких вычислительных процессов одновременно. В случаях, когда это требуется, можно разбивать задачи на мелкие части и управлять запуском мелких частей [1]. В этом случае вычислительные ресурсы будут использованы ещё менее эффективно из-за накладных расходов и простоя, но поведение других элементов страницы и программы-браузера будет более адекватным (в случае исполнения интенсивного кода другие процессы на странице блокируются, в некоторых случаях и на других страницах тоже).

Тем не менее можно ожидать, что рано или поздно такая возможность будет реализована. Рассмотрим способы конкурентного выполнения вычислительных потоков в рамках JavaScript.

CONCURRENT.THREAD

Эта библиотека, распространяемая по свободной лицензии, позволяет эмулировать многонитевое исполнение программы, разбивая программы на небольшие фрагменты. Конечно, это не может не сказаться на производительности.

Во-первых, расскажем о самой организации вычислений: для запуска параллельного процесса используется функция

Concurrent.Thread.create(function_var, function params…),

что само по себе достаточно удобно для тех, кто знаком, например, с нитями .NET. Если переписать определённый в начале данной статьи пример с использованием этой библиотеки, получим следующий код:

```
function startThread(num) {
     var l=0, dateObj=new Date(),
         startTime=dateObj.getTime();
     if (num < 1) return;
     console.log("Thread " + num + " started");
     Concurrent.Thread.create(startThread, num-1);
```

[4] Firefox 10.0.2, Safari 5.1, Google chrome 17, Opera 11, Internet Explorer 9

```
        while (dateObj.getTime()-startTime<3000) {
                dateObj=new Date();
                l+=Math.random();
        }
        console.log("Thread " + num + " ended (l=" + l
        + ")");
    }
    startThread(10);
    .
```

Кроме того, можно воспользоваться другим способом и обозначить фрагмент JavaScript-кода как многонитевой. Предыдущий пример привести к такому виду не получится, поскольку в нём требуется явное указание запуска различных потоков. Оформим синхронный HTTP-запрос как отдельный вычислительный поток [3]:

```
<script type="text/x-script.multithreaded-js">
    var   req   =   Concurrent.Thread.Http.get(url,   ["Accept",
"*"]);
    if (req.status == 200) {
        alert(req.responseText);
    } else {
        alert(req.statusText);
    }
 </script>
```

Выше мы упомянули понижение производительности при использовании этого подхода. Проведём несколько тестов для того, чтобы выяснить, насколько оно существенно. В качестве вычислительной задачи возьмём генерацию случайных чисел. Не будем приводить полный текст программы, поскольку это займёт достаточно места, используемая задача – генерация 10^7 псевдослучайных чисел. Такой цикл занимает примерно 100 мс в случае последовательного выполнения (в процессе которого блокируется работа браузера). В случае параллельного исполнения при помощи рассматриваемой библиотеки характерные времена исполнения отличаются на 2 порядка:

Можно отметить, что время отличается достаточно сильно, что делает невозможным использование этой библиотеки в вычислительных задачах, потому что потери времени масштаба двух порядков совершенно не оправданы. Тем не менее, у этого способа есть положительный момент: независимые вычисления не мешают друг другу. Это значит, что корреляции с количеством потоков в данном тесте не было обнаружено. Тестирование производилось на Google Chrome 17.

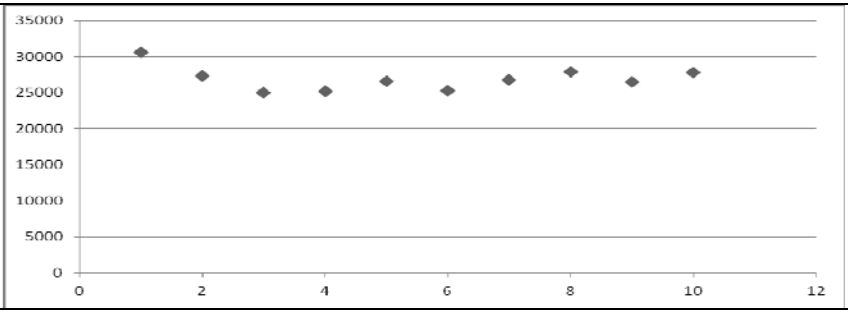

Рис. 1. Зависимость времени в миллисекундах от количества работающих потоков при использовании библиотеки Concurrent.Thread

MULTITHREADING

Другой подход [2], предложенный лабораторией компьютерных сетей École Polytechnique ,включает в себя отдельный компилятор и средства для отладки параллельных приложений на JavaScript. Подход изначально направлен не на получение высокопроизводительного кода, а на другую модель описания параллельных процессов. Эта модель также реализована в языке Synchronous C++ или sC++ [4]. Синхронизация в данном случае осуществляется посредством отправки сообщений. Реализованы операторы select, accept, waituntil, которые компилируются в обычный JavaScript-код при помощи Java-приложения. Программа разбивается на части между синхронизациями, которые не подвергаются изменениям, а для синхронизаций генерируется дополнительный код. Как уже упоминалось, для готовых программ создан отладчик, который позволяет проследить за синхронизацией.

В целом, такое решение больше подходит для учебных целей, а не для решения реальных задач, поскольку является достаточно громоздким и требует перекомпиляции при каждом изменении параллельного участка кода. Сам по себе оператор waituntil, который прерывает исполнение потока до указанного времени, возможно, и является тем средством, которого многим не хватало в JavaScript, но для удобства его использования следовало реализовать компилятор на самом языке JavaScript, как в прошлом случае. Приведём фрагмент программы на synchronous JavaScript:

```
process Channel(name) {
  var fifo = new Array(0)
  this.put = function(data) {
    fifo.push(data)
  }
  this.get = function(data) {
    return fifo.shift()
  }
  this.run = function() {
    for (;;) {
      select {
      case
        when (fifo.length>0)
        accept get
      case
        when (fifo.length<6)
        accept put
      }
      showChannel(fifo.length)
    }
  }
}
.
```

В примере описывается процесс, реализующий FiFo-очередь.

JQUERY DEFERREDS[5]

Достаточно популярная библиотека jQuery позволяет создавать объекты, которые помогают организовывать вычисления по готовности данных. Как правило, в JavaScript готовность данных означает, что закончился процесс их получения с сервера. Концепция не является сложной: создаётся объект, в который передаются функции, выполняемые в случае различных статусов при разрешении (в случае успеха, неуспеха, в любом случае). Как правило, такой объект возвращается функцией, которая выполняет загрузку данных, объект разрешается при помощи замыкания, контекста реального обработчика событий на контекст функции, вернувшей deferred-объект. Функции асинхронных запросов JQuery всегда возвращают deferred-объекты.

[5] http://api.jquery.com/category/deferred-object/

```
$.get("test.php").done(function() {
   alert("$.get succeeded"); });
```

В приведённом примере функция, выполняющая AJAX запрос типа get, возвращает deferred-объект, который позволяет назначить функцию, выполняемую в случае успешности запроса.

Также могут быть реализованы конвейеры из методов JavaScript, выполняемые асинхронно:

```
var defer = $.Deferred(),
    filtered = defer.pipe(function( value ) {
      return value * 2;
    });
defer.resolve( 5 );
filtered.done(function( value ) {
  alert( "Value is ( 2*5 = ) 10: " + value );
});
```

.

ВЫВОДЫ

Из рассмотренных способов организации параллельных процессов на JavaScript только Concurrent.Tread подходит для реализации вычислений. Возможно, разработчиками стандартов JavaScript будет выбран какой-то другой способ, но в связи с развитием браузерных приложений всё сложней обходить вниманием тот факт, что процессоры стали многоядерными.

СПИСОК ЛИТЕРАТУРЫ:

1. Edwards J. Multi-threading in JavaScript. — 2008. — Available at: http://www.sitepoint.com/multi-threading-javascript/ (accessed on 01.10.1012)
2. Petitpierre C. Multithreading for Javascript. Available at: http://ltiwww.epfl.ch/sJavascript/(accessed on 01.10.1012).
3. Maki D., Iwasaki H. JavaScript Multithread Framework for Asynchronous Processing: PhD thesis [online pdf document]. –15 p. – Available at: http://mirror.transact.net.au/sourceforge/j/project/js/jsthread/doc/thesis-en.pdf.
4. Petitpierre C. Synchronous C++, a Language for Interactive Applications // IEEE Computer. – 1998 – N. 9. – P. 65–72.