использование C и Ruby

Макс Лапшин. max@maxidoors.ru Сентябрь , 2006.

Существует по меньшей мере 4 способа «скрестить» код на Ruby и C
В этой заметке будет освещено несколько способов интеграции кода, написанного на C, с кодом на Ruby в целях расширения функциональности Ruby и оптимизации вычислений. Также будет описан еще ряд вопросов, возникающих при решении подобных задач.

Зачем писать на двух языках?

Первый вопрос, который может возникнуть — зачем такому высокоуровневому и удобному для программиста языку как Ruby какой-то C?

Во-первых, на C уже написано огромное количество библиотек. Некоторые уже видимо никогда не будут переписываться или заменяться — только безумец рискнёт переписывать glibc заново. Существуют библиотеки, написанные на C или C++, реализующие закрытые протоколы. Ситуация в индустрии сложилась так, что обычно на других языках такие библиотеки уже не пишут. C — это та самая «печка», от которой все пляшут.

Во-вторых, простой код на C при прочих равных может быть быстрее кода на руби. Это не всегда именно так, но C в большинстве случаев является хорошим резервом для ускорения программы. Например обработка бинарной строки в Ruby через unpack приводит к тому, что создаётся массив с соответствующим количеством элементов (записей в таблице объектов). Затем каждый элемент массива обрабатывается (поиски методов в хеш-таблицах), после чего все элементы запаковываются обратно (опять обращения к хеш-таблицам) и т.д. и т.п. Если алгоритм написать на C, то реализацией будет один метод, порождающий только одну новую строку, то есть в таблице объектов появится только одна новая запись. Сборщик мусора в Ruby пока что далёк от идеала по скорости, так что его лучше беречь от нескольких тысяч ненужных новых вхождений. При этом алгоритм на Ruby просто и естественно реализуется за минуту, написание того же самого на C занимает целый день, а ещё один день занимает отладка.

Рассмотрим несколько способов интеграции кода на C и Ruby.

Стратегии связывания

Можно выделить два подхода к написанию прослойки, связывающей C и Ruby:

- Можно писать обертку к какой-то библиотеке;
- можно делать функции-ускорители для Ruby.

Разница между этими подходами, как правило, заключается в том, что при создании обертки, на уровень C выносится основной объём логики. В случае создания ускорительных функций[1], они должны быть опциональны (т.е. отключаемы), и как правило логика локализуется внутри одной функции. Соответственно, для обоих подходов

[1] Надо отметить, что решение о переносе логики из Ruby в C должно приниматься только после проведения тщательного профилирования. Подробнее о профилировании — в следующих выпусках.

существуют различные инструменты, которые будут освещены ниже.

При создании обёрток к готовым библиотекам, можно придерживаться следующих подходов:

- Описывать на C максимальный объём логики, чтобы скомпилированная библиотека нуждалась в минимальном объёме дополнительного кода на Ruby. Подход хорош тем, что весь код локализован в одном языке.
- описывать на C минимально необходимую функциональность. Соответственно, достигается наибольшая гибкость кода, так как на Ruby писать проще. Появляются проблемы с расползанием кода по двум языкам, увеличивается использование памяти и растут накладные расходы на создание Ruby-объектов.

Использование Ruby API.

Чаще всего расширения к Ruby пишутся с использованием его API — самый естественный и самый гибкий метод. Почти вся функциональность Ruby создана с помощью собственного API. Суть подхода в том, что из кода на C программист регистрирует в интерпретаторе Ruby все необходимые классы, модули, методы и т.п.

Происходит это следующим образом: создается динамическая библиотека под названием, например, `udmsearch`, которая потом включается в Ruby вызовом `require 'udmsearch'`.

В этой библиотеке интерпретатор ищет функцию под названием `Init_udmsearch`[2]. Эта функция будет вызываться при включении модуля в Ruby. Функций для «отстыковки» модуля не предусмотрено из-за динамической природы языка. За время жизни модуля в объектной модели могут произойти необратимые изменения, в результате которых её состояние на момент запуска функции выгрузки (если бы она существовала) будет недетерминированно.

Вот типичный набор действий, который проводится в функции инициализации библиотеки: создание модуля в Ruby, создание в этом модуле нужных классов, регистрация их методов и т.п. Надо отметить, что этот инициализатор работает с теми же функциями, которые вызывает внутри себя Ruby при интерпретации текста на языке Ruby.

Рассмотрим использование этого API на примере работы с файловым API в glibc. Сделаем так, чтобы из Ruby уже существующий на диске файл можно было открыть на запись, дописать в него и закрыть.

Сначала надо подключить заголовочные файлы Ruby:

```
#include "ruby.h"
#include "rubyio.h"
#include "intern.h"
```

Класс, ответственный за файл, будет находиться в модуле `MyFile` и называться `File`. Для того, чтобы Ruby знал про этот модуль и класс, необходимо объявить статические (не на стеке) переменные типа `VALUE`:

```
VALUE mFile, cFile;
```

Все обращения к классу, к его методам (в том числе и статическим) будет происходить через эти переменные.

В теле функции `Init_myfile()` объявляется модуль `MyFile` и класс `File`. Следует заметить, что никакой связи между названием функции и названием модуля нет. Название функции совпадает с названием библиотеки, но в ней может объявляться много модулей. Объявление модуля происходит вызовом:

```
mFile = rb_define_module("MyFile");
```

---

[2] обратите внимание на то, как называется эта функция. Если библиотека создается компилятором C++, то функция `Init_udmsearch` должна быть защищена скобками:

```
extern "C" { void Init_udmsearch() {} }
```

Если модуль `MyFile` уже создан, то этот вызов вернёт ссылку на уже существующий объект (гибкость Ruby основана в том числе на том, что любые данные являются объектом). В том же модуле надо создать класс, наследующий от `Object`:

```
cFile = rb_define_class_under(mFile,
    “File”, rb_cObject);
```

Константа `rb_cObject` определена в заголовках и всегда существует. По умолчанию, любой класс является наследником от «просто объекта».

Не вдаваясь в тонкости процесса аллокации и инициализации объектов, объявим статический метод `new` класса `File`:

```
rb_define_singleton_method(cFile,
    “new”, f_open, 1);
```

Вот последовательность параметров:

1. Тот объект, на котором объявляется метод,
2. название метода,
3. функция (здесь `f_open`), которая будет обрабатывать этот метод,
4. количество аргументов. Могут быть значения от 0 до 15 (специальное ограничение), либо -1 — последнее значение отвечает за переменное количество аргументов.

Следует обратить внимание на то, как именно создается «статический» метод класса. На самом деле такого понятия в Ruby просто нет. Существуют методы класса и личные (singleton) методы объекта. Если у объекта есть его личный метод, то первым вызывается именно он, иначе идет поиск в таблице методов класса этого объекта, затем предка этого класса и так далее до объекта `Kernel`, который является предком всех классов. Таким образом, метод `MyFile::File.new` представляет собой личный (singleton) метод объекта `File`, принадлежащего к классу `Class`.

Опишем содержимое функции `f_open`:

```
VALUE f_open(VALUE self, VALUE name){
 FILE* f;
 Check_Type(name, T_STRING);

 f = fopen(RSTRING(name)->ptr, “w+”);
 if(!f) {
  rb_raise(rb_eStandardError,
   “Couldn’t open file”);
 }
 return Data_Wrap_Struct(self, 0,
  f_free, f);
}
```

Макрос `Check_Type` проверяет объект Ruby на соответствие его одному из нескольких внутренних типов. Строки, целые числа, дробные числа, символы, массивы, хеш-таблицы и просто объекты хранятся внутри по-разному. В случае если пришедший объект не проходит проверку, кидается исключение Ruby[3].

Категорически не рекомендуется жёстко привязываться к типу приходящего объекта, лучше вызывать метод `to_s` на параметре, но мы рассматриваем учебный пример и делаем упрощённый вариант с допущением. То же самое верно для массива. Не надо делать код более жёстким, чем нужно — достаточно просто проверить наличие метода `[]` на объекте.

После проверки типа входного параметра файл открывается на запись. Если точно известно, что тип пришедшего объекта — строка (это проверяется макросом), то показанным в примере образом можно добраться до содержимого строки. Длину строки можно также быстро выяснить: `RSTRING(s)->len`.

Функция `rb_raise` кидает исключение, которое сразу приводит к выходу из функции.

---

[3] Надо чётко понимать, что механизм исключений в Ruby не имеет ничего общего с механизмом исключений в C++. Подробнее смотри ниже.

Первым параметром может идти любой тип объекта. Если конструктор объекта, отвечающего за исключение, принимает не только строку, то формировать исключение следует другим способом.

Последняя конструкция в этой функции — создание нового объекта. Макрос `Data_Wrap_Struct` создаст новый экземпляр класса, переданного первым параметром (в данном случае `self` — именно класс, т.е. объект класса `Class`), вторым параметром передаётся функция-маркировщик для сборщика мусора (см. ниже), третьим параметром идет функция освобождения памяти (если указать вместо нее `NULL`, то память под структуру просто освободится через `free`), четвёртым параметром идёт указатель на данные создаваемого объекта. В этом примере специально указана функция освобождения памяти, потому что `FILE*` не освобождается через `free`. Опишем функцию `f_free`:

```
void f_free(FILE* f) {
}
```

Надо понимать, что на момент вызова функции освобождения памяти, все ссылки на объект уже уничтожены. Кидать исключения из этой функции нельзя, потому что исключение непредсказуемо возникнет в совершенно другой части программы во время создания нового объекта, не связанного с освобождаемым.

Осталось заявить на объекте `MyFile::File` два метода: `<<` и `close!`:

```
rb_define_method(cFile, “<<”,
   f_write, 1);
rb_define_method(cFile, “close!”,
   f_close, 0);
```

Опишем эти методы:

```
VALUE f_write(VALUE self, VALUE in){
  FILE* f;
  VALUE str;
  ID to_s = rb_intern(”to_s”);
  Data_Get_Struct(self, FILE, f);
  str = rb_funcall(in, to_s, in, 0);
  fwrite(RSTRING(str)->ptr,
    RSTRING(str)->len, 1, f);
  return self;
}
```

Каждой функции, отвечающей за метод объекта, всегда передаётся параметр `self`, отвечающий за сам объект. Его содержимое можно извлечь макросом `Data_Get_Struct`. В приведённом примере работа с входным параметром реализована правильнее. Параметр приводится к строке явно, вызовом метода `to_s`, на который обычно отвечает любой объект в Ruby. Надо отметить, что любая функция на C, отвечающая за метод Ruby-объекта, всегда обязана что-то возвращать. Если у программиста нет идей насчёт того, что функция должна возвращать, лучше вернуть сам объект, или, на крайний случай, — `Qnil;`

Метод `f_close` имеет похожую структуру и закрывает файл библиотечной функцией `fclose`. Учитывая, что сам объект не имеет ценности после его закрытия, логичнее из этого метода возвращать `Qnil`.

В чем же смысл возвращения ссылки на объект, а не статуса удачного завершения операции? Смысл в реализации так называемых fluent interfaces:

```
@f = MyFile::File.new “a.txt”
@f << str1 << str2 << str3
```

После написания такого кода, его надо скомпилировать. Для этого есть механизм `mkmf`. Надо создать файл, который обычно называется `extconf.rb`, со следующим содержимым:

```
require ‘mkmf’
create_makefile ‘myfile’
```

После выполнения этого скрипта будет создан Makefile с инструкциями по компиляции модуля. Обычно `extconf.rb` на порядок сложнее, но его структуру можно посмотреть в документации.

Описанная методика не является самой простой для тривиальных API, но в случае сложной логики на уровне C, это порой единственный рабочий механизм.

SWIG

SWIG — это генератор кода, который по переданному ему описанию интерфейса класса создает код для расширения. То есть на вход ему подаётся файл с описанием интерфейса, на выходе он генерирует текст на C.

SWIG достаточно неплохо догадывается о о том, как конвертировать типы Ruby в типы C и наоборот. Он даже умеет транслировать наследование C++ в Ruby и ловить исключения из C++ в Ruby. Если API библиотеки достаточно прост, то, возможно, пользоваться SWIG-ом будет удобно. В случае усложнения интерфейса использование SWIG-а перерастает в муторное написание большого объёма запутанного кода на C. Учитывая, что SWIG не предоставляет возможности вставки пользовательского кода в сгенерированный текст, использование этого инструмента может превратиться в мучение.

Важным плюсом SWIG-а является теоретическая простота портирования создаваемого модуля под другие языки, как-то Python, Perl и т.п. Надо отметить, что написание модуля для Python через его API является на порядок более сложной задачей по сравнению с написанием модуля для Ruby, поэтому имеет смысл помнить об этом преимуществе SWIG-а.

Лично я не видел ни одного изящного проекта, реализованного на SWIG-е, и неоднократно сталкивался с тем, что для эффективной реализации расширения требуется делать некоторые привязки к языку, для которого делается расширение, и к внутренней структуре его данных. Например чтобы избежать лишнего копирования строк (см. ниже)

Рассмотрим пример работы со SWIG. Сам генератор принимает файлы с описанием интерфейса. В этом интерфейсе указан модуль, в который будут генерироваться функции, указываются эти функции, правила преобразования типов данных из C в Ruby и наоборот, классы и т.п. Список возможностей SWIG-а поражает: поддержка полиморфизма C++, генерация предикатных методов (со знаком вопроса на конце), работа с исключениями и т.д.[4]

```
%module "MyFile"

FILE* fopen(char *name, char* mode);
size_t fwrite(void *buf, size_t size,
   size_t count, FILE *f);
```

Это содержимое простого интерфейсного файла. Соберем его (используя SWIG версии не ниже 1.3.29):

```
swig -ruby my_file.i
```

Теперь это надо скомпилировать (подробности компиляции — см. Makefile из предыдущего раздела). После этого можно загружать:

```
require ‘MyFile’
f = MyFile::fopen(”my_file.i”,”r”)
```

Лично мне не ясно, можно ли сделать удобный объектный интерфейс с помощью SWIG. В документации не приводилось хороших примеров. Если вам что-нибудь об этом известно — в конце статьи описано, как со мной связаться и рассказать об этом.

DL

Этот механизм является самым примитивным из описываемых. DL — просто обёртка над функциональностью загрузки динамических библиотек. Этот модуль позволяет загрузить нужную библиотеку, затем на Ruby описываются функции из этой библиотеки (по именам) с параметрами, и к этим функциям уже можно обращаться. Надо отметить, что этот API требует детальнейшего знания об интерфейсе библиотеки (который может меняться), да и

---

[4] Сугубо личное мнение автора заключается в том, что при всей функциональности SWIG-а, изящных решений при его использовании не получается. К созданному с помощью SWIG расширению всё равно придётся на Ruby писать объектную обёртку. Чаще всего логика из C не отображается напрямую в Ruby. Если же писать обёртку на уровне C, то будет потерян основной плюс SWIG-а: переносимость между скриптовыми языками. Впрочем, это мнение не подтверждено практикой, так что лучше убедиться в его правильности или неправильности самому.

результирующий код на Ruby изящным не назовешь.

Рассмотрим пример работы с этим API:

```
require ‘dl’
module LIBC
 LIB = DL.dlopen(”/lib/libc.so”)
 SYM[:fopen] = LIB[’fopen’,’PSS’]
 SYM[:fclose]= LIB[’fclose’,’0P’]
def fopen(name, mode)
 r,rs = SYM[:fopen].call(name,mode)
 r
end
def fclose(ptr)
 SYM[:fclose].call(name,mode)
end
end
```

Таким образом, для разумной работы с функциями в C требуется 3 (!) уровня абстракции: первый — это сам механизм DL, способный маршалить данные и находить нужные функции, второй — описание всех этих функций в каком-либо модуле, третий уровень — создание объектной надстройки над этим описанием.

Ценность такого механизма я обсуждать не хочу, поскольку в каких-то случаях это, возможно, и применимый интерфейс. Точно можно сказать, что при его использовании технически невозможно работать с библиотекой на C++. Важно также то, что добраться до полей структур невозможно, как, впрочем, и невозможно передавать сами структуры.

Rubyinline

Это самый новый и интересный механизм. Рассмотрим тестовый пример. Пусть существует файл размером в мегабайт, задача состоит в подсчёте суммы всех чётных байт. Если решать эту задачу только средствами Ruby, то, в зависимости от алгоритма, будет создано порядка миллиона объектов, несколько раз запущен сборщик мусора и скорее всего будет забита память.

Учитывая, что особой алгоритмической сложности эта задача не представляет, имеет смысл решить её на уровне C. Выше уже было приведено несколько механизмов такого опускания логики вниз, но все они страдают некоторой громоздкостью в применении к данному случаю.

Есть другой вариант решения нашей задачи — Rubyinline. Суть его в том, что текст на C находится прямо в методе класса, записанный в строку. При обращении к этому методу происходит компиляция и линковка получившегося кода на ходу.

Рассмотрим решение задачи:

```
class MyParser
inline do |builder|
 builder.c “
 long counter(char* name){
 FILE* f = fopen(name, “r”);
 unsigned int i, count = 0;
 unsigned char b;
 for(i = 0; !feof(f); i++) {
  fread(&b, sizeof(b), 1, f);
  if(i%2 == 0) {
 count += b;
 }
 fclose(f);
 return count;
 }
}”
end
end
MyParser.new.counter(”a.dat”)
```

Метод `inline` добавит в класс `MyParser` новый метод, совпадающий по сигнатуре и названию с функцией внутри текста. Обратите внимание на то, что `self` не передаётся. Можно также отметить, что этот механизм не позволит удобно делать обертки вокруг структур, запаковываемых в Ruby-объекты.

Rubyinline позволяет работать с C++. Возможности работать со структурами, увы, отсутствуют.

Надо отметить, что при использовании этого механизма в программе на Ruby легко могут возникать непереносимые конструкции, связанные с опциями компилятора или заголовочных файлов. Этот механизм также требует наличие компилятора, что делает очень затруднительным его использование с Windows[5].

Работа с памятью и сборка мусора

Работа с памятью в C характеризуется одной фразой: «головная боль». Ниже перечислены наиболее частые сценарии возникновения утечек или ошибочного обращения к памяти:

- В начале программы некоторой функции сделали `malloc`, а ниже забыли сделать `free`, или вызвали `return` до освобождения памяти;
- запутались с ответственностью за память. «Заповедь» гласит: ресурсы освобождаются на том же уровне, на котором и захватываются, но иногда это катастрофически неудобно. Функция `readline` возвращает выделенную динамически область памяти, потому что программисту неизвестно, сколько символов введет пользователь, и фиксированное количество памяти заранее выделить невозможно. Значит, надо об этом помнить. Все эти проблемы могут возникнуть при написании расширения.

Ruby позволяет решать эти проблемы с помощью своего сборщика мусора, однако надо понимать принципы его работы чтобы не наломать дров. При создании любого объекта (программист получает тип `VALUE`) происходит его регистрация в общей таблице объектов. При запуске сборщика мусора происходит обход графа связей объектов, начиная с глобальных переменных и дальше по ссылкам с них вниз.

Если созданный в расширении объект зависит от наличия другого объекта, но класть ссылку на него в переменную объекта (`rb_iv_set`) не хочется, то обязательно надо пометить нужный объект при обходе. Происходит это следующим образом: объект `Data_Wrap_Struct` вторым параметром принимает функцию `mark`. Если сборщик мусора посчитает данный объект нужным, он его и пометит как нужный и вызовет на нём эту функцию (если программист не указал вместо нее значение 0).

Защитить данные от зачистки можно двумя способами[6]. Во-первых, можно поместить копию VALUE (это просто указатель) в структуру класса и при вызове указанной функции `mark` вызывать функцию:

```
rb_gc_mark(self_ptr->required_obj);
```

Именно так и работает, например, хеш-таблица или массив. При каждом вызове функции маркировки они помечают, как живые все объекты внутри контейнеров.

Второй вариант — положить ссылку на необходимый объект в переменную собственного объекта:

```
rb_iv_set(self, “@obj”, obj);
```

Напоминаю, что при запуске сборщика мусора, тот проходит по стеку в поисках объектов, которые созданы, но ссылки на которые еще не проставлены. Если в стеке есть ссылка на объект, он не будет уничтожен.

Работа с данными Ruby-объектов

Любое взаимодействие с объектом снаружи происходит через посылку ему сообщений. Посылка сообщений из C реализуется функцией `rb_funcall(object, message, count,…)`

---

[5] Информация для людей, не знающих, что такое Windows. Это морально и технически устаревшая операционная система. Из важных особенностей: не имеет в штатной поставке компилятора.

[6] Есть еще вариант пометки переменных как глобальных, но я его принципиально не хочу озвучивать, как грязный. Лучше делать переменные класса.

Например, создание нового файла может выглядеть так:

```
VALUE f;
f= rb_funcall(cFile,rb_intern(”new”),
1, rb_str_new2(”a.txt”));
```

Объекту `cFile` посылается сообщение `new`, параметризованное одним аргументом.

Работа с объектом изнутри подразумевает доступ к членам объекта и членам его класса. Для этого есть семейство функций

`rb_iv_get, rb_iv_set` и `rb_cv_get, rb_cv_set`.

Подробное описание этих функций есть в документации, но надо отметить одну важную особенность. Собственная переменная объекта, являющегося классом для другого объекта — совершенно не то же самое, что переменная класса для объекта. Иначе говоря:

```
rb_cv_get(f,rb_intern(”@@a”));
rb_iv_get(cFile, rb_intern(”@a”));
```

— обращения к разным переменным. Более подробное объяснение в следующих выпусках.

Работа с внутренней структурой Ruby-объектов напрямую

Как было сказано в одном из предыдущих разделов, некоторые из стандартных типов объектов имеют чёткую структуру, описанную в заголовочных файлах. Если вас не пугают проблемы, которые могут возникнуть из-за смены внутренней структуры, можно к стандартным типам обращаться напрямую. Иногда эта, в целом порочная, практика очень сокращает объем копируемой впустую памяти. Рассмотрим два стандартных типа, доступ напрямую к которым имеет смысл.

Строки в Ruby имеют внутренний код типа `T_STRING`[7]. К ним можно обратиться напрямую, приведя `VALUE` к указателю на структуру строки: `RSTRING(str)->ptr`. Вот пример целесообразного использования этого подхода:

```
VALUE f_read(VALUE self, VALUE size){
  int _size = NUM2INT(size);
  FILE* f;
  VALUE str = rb_str_buf_new(_size);
  Data_Get_Struct(self, FILE, f);
  _size = fread(
    RSTRING(str)->ptr, _size, 1, f);
  RSTRING(str)->len = _size;
  RSTRING(str)->ptr[_size] = ‘\0’;
  return str;
}
```

Как видно, такой «нечестный»[8] прием позволяет избежать ненужного копирования данных. Работая со строками таким образом, не надо забывать про дописывание нулевого байта в конец строки. Для Ruby он совершенно не нужен, но любая библиотечная функция на C будет не рада, не найдя его, о чем известит `SEGFAULT`.

Работа с массивами организована аналогично:

```
int i;
int total_size = 0;
for(i=0;i<RARRAY(a)->len;i++) {
 VALUE str = RARRAY(a)->ptr[i];
 total_size += RSTRING(str)->len;
}
```

В этом примере подсчитывается суммарная длина строк, записанных в массив. Если вдруг в этом массиве окажутся объекты, не являющиеся строками, вы даже не получите исключения. То есть такой код допустим в том и только в том случае, если скорость очень, **очень** критична (недопустимо многократное переключение

---

[7] Надо понимать, что с точки зрения кода на Ruby, строки — точно такой же тип данных, как и любой пользовательский. Но при работе с ними на уровне C, они помечаются тегом `T_STRING`, в то время, как любой пользовательский объект помечается тегом `T_OBJECT`.

[8] Детальное описание «честных» приемов работы со строками можно прочитать в документации, ссылки на которую есть в конце статьи.

контекста или проверка типа) и совершенно точно известно, что в массиве будут строки.

По личному опыту могу заявить о порочности вышеприведённой практики. Такой код является не просто оптимизацией, а «запилом». Он обязан быть отключаемым со стороны внешнего для класса клиента, причем достаточно просто. Если мне потребуется подсунуть не массив, а итератор, читающий из сетевого сокета хендлеры к прокси-объектам строк, алгоритм принципиально не изменится, но этот код на C перестанет работать.

Многонитевой доступ

Работа с нитями в Ruby — это отдельная больная тема. Ситуация такова, что поддержка нормальных системных нитей появится только в ruby версии 2.0, до которого на момент написания статьи по меньшей мере ещё год-полтора. В ruby 1.8 нити выполнены, как внутреннее понятие интерпретатора, существующее в одной реальной нити операционной системы. Любая операция, выполняемая на C, с точки зрения Ruby является атомарной. Если же вы хотите воспользоваться работой с настоящими нитями, то лучше писать так, чтобы интерпретатор ruby находился в одной нити, а остальные нити не были бы связаны с ruby. Впрочем, я такие опыты не производил.

Вот как выглядит работа с псевдонитями в Ruby. Пусть происходит сортировка массива. Массив на это время блокируется. Блокировка происходит не настоящая, не атомарная, ну а в коде на C просто выставляется один флаг. Наверняка известно лишь, что выставление и проверка этого флага не может быть прервано на выполнение другого кода на ruby в рамках того же интерпретатора. Однако, при вызове операции сравнения над объектами в массиве, управление получит интерпретатор, который может переключить свою внутреннюю нить на другую, в которой будет предпринята попытка модификации массива. В результате вторая нить будет заблокирована, но не средствами настоящей, атомарной, блокировки библиотеки `pthreads`, а средствами интерпретатора ruby, ведущего свой список нитей.

Такой подход проще в реализации тем, что тут отсутствуют проблемы с конкурентным доступом на уровне C, нити переключаются быстрее по сравнению с настоящими, но при этом есть проблемы совместимости с настоящими нитями. Эту проблему в ruby только предстоит решить.

Компиляция под Windows

Как было сказано выше в сносках, в Windows вообще есть серьезные проблемы с компилятором. Компилятор от Microsoft работает в несколько раз медленнее, чем gcc, а использование gcc под Windows далеко не так тривиально, как в случае с Unix. Лично мне не удалось добиться работоспособности своего кода под Windows: проблема была в том, что последний компилятор VC 2005 (version 8) собирал код, несовместимый с другим кодом, собранным VS .NET 2003 (version 7). Кому-то удача улыбается больше. Если вам улыбнётся — черкните пару строк.

Список ссылок

http://www.google.com/search?q=extending+ruby

http://www.rubycentral.com/book/ext_ruby.html

http://ttsky.net/ruby/ruby-dl.html

http://zenspider.com/ZSS/Products/RubyInline/

Автор

Макс Лапшин

max@maxidoors.ru

http://maxidoors.ru/