Ruby/DL

Takaaki Tateishi June 17th Heisei 14

Hastily translated by Benjamin Peterson, ben@jbrowse.com
2 unclear areas are marked with (???), please let me know if you figure them out

Contents

1 Tutorial

In most OSes, there are object files called by programs at run time, which are called DLLs or shared libraries. In this document, these object files are called by the general name of 'system libraries'. This section will introduce the calling of functions defined in such libraries (hereinafter 'library functions') using import.rb, which is part of Ruby/DL. It is also possible to use library functions via the basic example-level functions defined in the DL module. This method will be explained in the next 2 sections. In import.rb, the DL::Importable module is defined. Using this module, library functions can be used from Ruby in a visually clear way.

1.1 Importing library functions

Using import.rb, in order to call library functions from Ruby, you must take the following 3 steps.

- Define a module for the system library
- Load the library you want to use
- Make the library functions you want to use accessible from Ruby

The module for the system library must extend the DL::Importable module. This is because DL::Importable defines methods to load and use the library. First, we will describe the definition of this module. For instance, if you want to use the C library 'libc.so', the module definition is as follows:

```
require 'dl/import'
module LIBC
extend DL::Importable
end
```

Here, something else could be used instead of the name LIBC, but please be sure to extend it from DL::Importable. To load the system library, the dlload function from this LIBC module will be used. Further, the library functions will be made usable from Ruby via the 'extern' function. This process of making library functions available to Ruby is called 'importing'.

```
require 'dl/import'
module LIBC
extend DL::Importable
dlload "libc.so"
extern "int strlen(const char *)"
end
```

In the example above, the strlen function found in libc.so has been made accessible. strlen() is one of the standard C library functions, returning the length of a string. It is possible to specify multiple system libraries as the

arguments to dlload. The extern function must be given a C function prototype. However, because not every type is necessarily defined, it may be necessary to define each undefined type that you want to use using the method given in section 1.1.2. The function can be called from Ruby when using the LIBC module as follows:

```
LIBC.strlen("abc") # => 3
```

The return value should be 3. In ruby, a method name cannot begin with a capital letter, but in C it can. Because of this, if the imported function's name begins with a capital letter, it is automatically replaced with a lower case one. For example, the library function GetString() is accessed from Ruby via the getString() method.

1.2 Type definitions

Ruby/DL understands ordinary C types, but types specific to a particular environment or library are mostly not supported. Accordingly, when describing function prototypes, it is necessary to define all the unsupported types in advance, in the module. For instance, size_t is not defined in Ruby/DL. To define it as an unsigned int, add a line like the following:

```
typealias("size_t", "unsigned int")
```

You could also leave out the parentheses and write it like so:

```
typealias "size_t", "unsigned int"
```

1.3 Arrays, pointer type data, and symbols

1.4 Callbacks

A few library functions require callback functions. For instance, the C library function qsort() requires a pointer to a function that will be used to compare elements.

```
void qsort(
    void *base,
    size_t nmemb,
    size_t size,
    int (*compar)(const void *, const void *)
)
```

First, put the following definition in LIBC to allow qsort() to be used from Ruby:

```
require 'dl/import'
module LIBC
extend DL::Importable
dlload "libc.so"
typealias ("size_t", "unsigned int")
extern "void *qsort(size_t, size_t, void*)"
end
```

Because the function type "int (*)(...)" is not understood by Ruby/DL, it will be defined as a pointer type. Next, the callback function is defined in Ruby.

```
module LIBC
def my_compare(ptr1, ptr2)
   ptr1.ptr.to_s <=> ptr2.ptr.to_s
end
MY_COMPARE = callback "int my_compare(char**, char**)"
end
```

First, the Ruby method my_compare is defined. This method takes 2 pointers to strings and compares them. Next, the "callback" method is used to make the method we have defined work with C. The argument is a function prototype describing a function with the same name as the method we defined. The return value of "callback" is a Symbol object representing the C symbol.

```
ary = ["a","c","b"]
ptr = ary.to_ptr
LIBC.qsort(ptr, ary.length, LIBC::MY_COMPARE)
```

1.5 Reference arguments

2 Ruby/DL details

In this section, using a function in a C library as an example, we aim to briefly explain the usage of Ruby/DL. In most Unix-like operating systems, I think the standard C library is in /library/libc.so or /usr/lib/libc.so, but here we well use /usr/lib/libc.so. The reader should have a basic knowledge of C -- at least, I think it will be hard to understand this part without any knowledge of pointers.

2.1 How to call functions

We will take the C library functions isdigit() and atoi() as our examples. First, in order to use the libc.so library, we use the DL.dlopen module function.

```
require 'dl'
libc = DL.dlopen('/usr/lib/libc.so')
```

DL.dlopen(path) opens the library denoted by *path* and returns the handle as an instance of the DL::Handle class. In the example above, the variable 'libc' has been assigned a Handle object.
Next, we extract pointers to the C library functions atoi() and isdigit(). To obtain the function pointers, either the Handle#sym method, or the shortened form Handle#[], can be used.

```
atoi = libc['atoi', 'IS']
isdigit = libc['isdigit', 'II']
```

In Ruby/DL, these function pointers are handled by the DL::Symbol class. Thus, the atoi and isdigit variables denote instances of DL::Symbol. In the first argument, the desired function's symbol is specified as a string. Then in the second argument, the type of the function is specified. The type is represented as a string composed of characters known as type specifiers. The first character represents the function's return type, and the *k*-th character represents the *(k-1)*-th argument's type. In the case of atoi, the return type is 'I', indicationg a C 'int', while 'S' indicates a 'const char *'. When the return type is not necessary, the first character can be '0' (zero). There is a detailed description of type specifiers in section 3.5.

To use the function pointer thus obtained to actually call a function, the 'call' method is used.

```
r1 = isdigit.call(?1) # => [2048, [49]]
r2 = isdigit.call(?a) # => [0, [97]]
r3 = atoi.call('10') # => [10, ["10"]]
```

Again, instead of "call", you can use brackets.

```
r1 = isdigit[?1] # => [2048, [49]]
r2 = isdigit[?a] # => [0, [97]]
r3 = atoi['10'] # => [10, ["10"]]
```

r1, r2 and r3 contain an array into which a list of the values returned after the function has been called are substituted. The various comments written on the right are examples of the values returned by Symbol#call/[].
Supposing you want to define a str2int function that takes a string of digits and converts it to an integer value using atoi, using the process flow described above; it can be done as follows:

```
require 'dl'
```

```
module LIBC
LIB = DL.dlopen('/usr/lib/libc.so')
SYM = {
   :atoi => LIB['atoi', 'IS']
   :isdigit => LIB['isdigit', 'II']
}
def atoi(str)
   r,rs = SYM[:atoi].call(str)
   return r
end
def isdigit(c)
   r,rs = SYM[:isdigit].call(c)
   return (r == 0)
end
end

include LIBC
def str2int(str)
   str.each_byte{|c|
      if( LIBC.isdigit(c) )
      return nil
      end
   }
   return LIBC.atoi(str)
end
```

In this example, by the wondrous power of the LIBC module, functionality offered by libc.so can now be provided by Ruby. The constant LIB holds a library Handle object; SYM is defined as a hash for holding Symbol objects. The module functions atoi and isdigit expose the library functions of the same names.

2.2 mutable arguments

With C functions, the values of the arguments may change during the execution of the function. For instance, strcat() appends the string given in the second argument to that given in the first. The first string is therefore not the same after the execution of the function. In order to handle this kind of 'mutable' function, the type specifier 's' (lower case) is used. Adding the strcat() function to our LIBC module would look like this:

```
module LIBC
SYM[:strcat] = LIB['strcat', 'SsS']
def strcat(str1,str2)
```

```
    r,rs =
    SYM[:strcat].call("#{str1}\0#{str2}",str2)
    return rs[0]
end
end
```

What you have to take notice of here is that the first argument of SYM[:strcat].call does not just recieve simply str1, but "#{str1}\0#{str2}". This is because, when strcat() is used in C, the first argument must refer to a buffer that has enough space to hold the completed string after the function has finished. With the inputs we are giving to this module function, "abc" and "def", the result will be to return "abcdef".

```
include LIBC
LIBC.strcat("abc", "def") # => "abcdef"
```

2.3 How to use pointers

When writing a program in C, there are a lot of places where you use pointers. For instance, you use fopen() to open a file. The function takes 2 arguments. The first argument is the file name as a string. The second is the mode in which to open the file, as a string. The return value is a pointer to a FILE structure. Thus, the type of fopen() is represented as 'PSS' in Ruby/DL. The type specifier 'P' represents a pointer type. LIBC module functions to handle fopen(), fclose(), and fgetc() in Ruby can be defined as follows:

```
module LIBC
SYM[:fopen] = LIB['fopen', 'PSS']
SYM[:fclose] = LIB['fclose', '0P']
SYM[:fgetc] = LIB['fgetc', 'lP']
def fopen(filename, mode)
    r,rs = SYM[:fopen].call(filename, mode)
    return r
end
def fclose(ptr)
    SYM[:fclose].call(ptr)
    return nil
end
def fgetc(ptr)
    r,rs = SYM[:fgetc].call(ptr)
    return r
end
end
```

LIBC.fopen gets you a pointer depending on the file name and mode specified

when you open the file. The file pointer is an instance of DL::PtrData. The PtrData class is the class for handling pointers from Ruby, and defines methods for managing pointers. A function defined in LIBC to use these functions to output the contents of a file would go like this:

```
include LIBC
def print_file(filename)
fp = LIBC.fopen(filename, "r")
if( !fp ) # -- (note 1)
return nil
end
while( (c = LIBC.fgetc(fp)) > 0 )
print(c.chr)
end
LIBC.fclose(fp)
end
```

C's 'NULL' is treated as 'nil' in Ruby. Thus, to see if file opening has failed, check the value (note 1) returned from LIBC.fopen to see whether it is nil or not. In Ruby/DL, when 'P' or 'p' is given as a type specifier, at the time when the function is called you can check whether the objects given as arguments are PtrData or not. If an argument is not a PtrData, you can convert it to a PtrData by using the to_ptr method. Using this technique, C arrays and strings can be handled alike, as pointers. After 'require'-ing the Ruby/DL module, the methods String#to_ptr and Array#to_ptr are defined, so, for instance, you could use 'P' and 'p' instead of the 'S' and 's' type specifiers. For example, the C library function strlen() can be handled in the LIBC module as follows:

```
module LIBC
SYM[:strlen] = LIB['strlen', 'IP']
def strlen(str)
   r,rs = SYM[:strlen].call(str)
   return r
end
end
```

2.4 How to use arrays

To pass a Ruby array to a C function, the type specifier 'A' or 'a' are used. For instance, the C library function qsort() from glibc2 has this prototype:

```
void qsort(
   void *base,
   size_t nmemb,
   size_t size,
```

```
   int (*compare)(const void *, const void *)
)
```

Thus, the Ruby/DL type specifiers '0aIIP' would be used. qsort() is a general-purpose sort function, but on this occasion we will only consider sorting an array of strings. Since we have already described how to define the function for qsort()'s 'compare' argument in Ruby in section 2.6, here we will use a function that we have put in a library ourselves. This is a function to compare 2 strings:

```
#include
int mystrcmp(char **str1, char **str2)
{
   return strcmp(*str1, *str2);
}
```

We will call the library which contains this function 'libmy.so'. Now, to use this library we will have to make a Ruby module like this:

```
module LIBMY
LIB = DL.dlopen('libmy.so')
SYM = {:mystrcmp => LIB['mystrcmp', 'IPP']}
end
```

If we do this, then we can obtain a pointer to mystrcmp() in the library with LIBMY.SYM[:mystrcmp]. Lastly, we define a LIBC module function for using qsort() thusly:

```
module LIBC
SYM[:qsort] = LIB['qsort', '0aIIP']
def qsort(ary, comp)
   len = ary.length
   r,rs = SYM[:qsort].call(ary,
      len,
      DL.sizeof('P'),
      comp)
   return rs[0].to_a('S', len)
end
end
```

rs[0] is the value of the first argument after the function invocation, but because the type specifier 'a' was used, the value that can be obtained on the Ruby side is a PtrData object. Therefore, rs[0] is converted to an array with the to_a method. When that is done, you specify how to convert the pointer by supplying 2 arguments. In the first argument, you pass the type of the array elements; in the second, the number of elements in the array. In this case, the elements are strings, so 'S' is passed in the first argument. As the number of

elements is the same as before the invocation, the value that was obtained earlier is passed in argument 2. In this way, LIBC.qsort can handle an array of strings and compare them with the C function given in 'comp', returning the result again as an array of strings. Using the LIBC and LIBMY modules, we can define a Ruby method that sorts a string array using mystrcmp() like this:

```
include LIBC
include LIBMY
def strsort(ary)
   LIBC.qsort(ary, LIBMY[:mystrcmp])
end
```

2.5 How to use structures

Is there are many situation in which structures are defined in C, it is possible to create new types. For instance, the timeval structure used by gettimeofday() is defined as follows:

```
struct timeval {
   long tv_sec;
   long tv_usec;
};
```

In order to reserve memory space for a structure like this, either DL.malloc or Array#pack can be used. DL.malloc takes the amount of memory needed as an argument. It returns a PtrData object that indicates the memory range allocated. In such a case, you can use DL.sizeof to get the size of a given structure. DL.sizeof can also calculate the amount of memory required for a structure type. It takes alignment into account. The two lines below may appear to be doing the same thing, but in fact (1) displays the size of a structure that contains a char and a long, and (2) simply returns the size of a char type plus the size of a long type.

```
ptr = DL.malloc(DL.sizeof('CL')) # (1)
ptr = DL.malloc(DL.sizeof('C') + DL.sizeof('L')) # (2)
```

When using Array#pack, having once made a string in Ruby you can convert it to a PtrData with to_ptr:

```
ptr = [0,0].pack('ll').to_ptr
```

Using pack, each element can be initialized at once.

When a structure is represented by a PtrData object, the various elements can be accessed by supplying an offset. For instance, with the following code you

can find the elements corresponding to tv_sec and tv_usec:

```
sec = ptr[0].to_i
usec = ptr[DL.sizeof('L')].to_i
```

We now present the C code for comparison with the equivalent Ruby code:

```
#include
#include
#include
int main()
{
   void *ptr;
   long sec, usec;
   /* ptr = [0,0].pack('ll').to_ptr */
   ptr = (void*)malloc(sizeof(struct timeval));
   ((struct timeval *)ptr)->tv_sec = 10;
   ((struct timeval *)ptr)->tv_usec = 100;
   /* sec = ptr[0].to_i */
   /* sec = (ptr + 0).ptr.to_i */
   sec = *(long*)(ptr + 0);
   /* usec = ptr[DL.sizeof('L')].to_i */
   /* usec = (ptr + DL.sizeof('L')).ptr.to_i */
   usec = *(long*)(ptr + sizeof(long));
   printf("sec = %ld, usec = %ld\n", sec, usec);
   exit(0);
}
```

However, normally, starting with the base address of the structure and applying an offset followed by a cast is not what you do to access structure members. In C, the code looks something like this:

```
#include
#include
#include
int main()
{
   struct timeval *ptr;
   long sec, usec;
```

Depending on the C compiler there might be a warning, or it might not go through at all.

```
   ptr = (struct timeval *)malloc(sizeof(struct timeval));
   ptr->tv_sec = 10;
   ptr->tv_usec = 100;
```

```
    sec = ptr->tv_sec;
    usec = ptr->tv_usec;
    printf("sec = %ld, usec = %ld\n", sec, usec);
    exit(0);
}
```

This is possible because it is understood that the variable called ptr has a value which is a timeval structure. In Ruby/DL, the struct! method is provided to fill in the individual elements of a structure. The first argument of this method is a collection of the types of the structure members. The second argument consists of symbols to be used as keys in accessing the members.

```
ptr = DL.malloc(DL.sizeof('LL'))
ptr.struct!('LL', :tv_sec, :tv_usec)
ptr[:tv_sec] = 10
ptr[:tv_usec] = 100
sec = ptr[:tv_sec]
usec = ptr[:tv_usec]
```

Again, in the same way as the struct! method is provided for members of structures, there is a union! method for members of unions.

2.6 How to use callbacks

In C, function pointers can be passed as arguments. For instance, in the qsort() method mentioned before, the comparison function needed for sorting is supplied as an argument. In the example just now, we used a function already defined in C, but this time we will define the function on the Ruby side. Firstly, the function would look like this in C:

```
#include
int mystrcmp(char **str1, char **str2)
{
    return strcmp(*str1, *str2);
}
```

To define the callback function in Ruby/DL, use the DL.callback module function.

```
cb = DL.callback('IPP'){|ptr1, ptr2|
    str1 = ptr1.ptr.to_s
    str2 = ptr2.ptr.to_s
    str1 <=> str2
}
```

The type specifiers for the function are provided as arguments; in this case, the function takes 2 pointer types and returns an integer. However, in Ruby callbacks can only be defined so far as they are countable(???). ptr1.pointer corresponds to C's '@str1'. The to_s method is used to convert the data already obtained from the pointer into a string. DL.callback's return value is a Symbol object. This can be used with the qsort method we have already defined in the LIBC module.

```
include LIBC
qsort(["c","a","b"], cb1)
```

2.7 Defining the 'free' function

In C the usual system is to allocate memory with malloc() and release it with free(). For instance, if the string "abc" is converted to a PtrData type, the result is like this:

```
ptr = "abc".to_ptr
# => #<DL::PtrData:0x8096f48
# ptr=0x8096788
# free=0x402e28c8>
```

'ptr' is a pointer value representing this PtrData. 'free' is a function that will be run on this PtrData object when it is recycled by the GC; in this case free() will be called. A Symbol object for free() is pre-defined in Ruby/DL as DL::FREE.

```
DL::FREE
# => #<DL::Symbol:0x8096658
# func=0x402e28c8
# 'void free(void *);'>
```

The value denoted by 'func' is a pointer to this function.

The PtrData object's 'free' member can be changed by the user. For instance, to define the 'free' function of the PtrData object 'ptr' as the function 'sym', you can do 'ptr.sym = sym'. (??? surely this should be ptr.free = sym)

3 Ruby/DL Manual

3.1 DL Module

3.2 DL::Handle class

3.3 DL::Symbol class

3.4 DL::PtrData class

3.5 Type Specifiers

3.6 DL::MemorySpace module

4 Interoperability with other extension libraries.