Programming with Haiku

Lesson 15

Written by DarkWyrm

Our Own File Types

While there are a great many different kinds of files already available which cover everything from pictures to video to e-mail, sometimes it is necessary for our programs to use their own specialized types of files. While it's possible to just create a file with some non-standard extension, actually creating a useful, full-blown file type in Haiku involves a little more work. None of it is difficult, but it does require a piecing together some odds and ends from different parts of the Storage Kit to make it all work well.

The process for making our own file type has a few basic steps: determine the MIME type and extension(s), design an icon for the type, and installing it into the MIME database. Once it's installed, there isn't much needed to use your new file type, either.

Creating the New Type

First, we need to think up a little information before we can create our new file type. Ask yourself three questions:

1. What is the extension(s) for my new type?
2. What is the MIME type for my new type?
3. What will the icon look like?

For our purposes here, we'll say we're creating a new kind of text document for a word processor called MyWrite. It would make sense to use an extension that is obviously tied to our program, so we'll use a 4-letter extension: mywr.

MIME types aren't complicated, but they do require a little thought. The supertype for the document must be determined. MyWrite files are text documents, so they have the `text` supertype. Choices for other files include `audio`, `video`, `image`, and `application`. The application type is generally used for formats which don't fit into another supertype. Compressed archives, such as 7-Zip and BZip2 fit into this category. Once the new type's supertype is set, the rest of the type is constructed. Non-standard MIME types start with `"x-"`. Vendor-specific types start with `"vnd."`. For MyWrite, the full MIME type will be `text/x-mywrite`.

For some developers, the icon is the hardest part because they write code, not draw pretty pictures. A type's icon doesn't have to be fancy, although it does reflect better on your program if it looks reasonably professional. Haiku's icon tool of choice is Icon-O-Matic, which happens to be bundled in with the rest of the operating system. For the sake of focus, we won't go into the details of creating an icon with Icon-O-Matic – that will be another day's lesson. We'll just continue on and assume that you have an icon that will work.

To install our new MIME type we will make use of the BMimeType class. It is the official interface with Haiku's MIME database. While we could use the FileTypes preferences application, we need to be able to create the type programmatically. Here's some code that will explain how to install a basic type.

```
#include <Mime.h>       // The header for BMimeType

// A header which holds the icon data converted into C++ code -- more on
// this afterward.
```

```
#include "IconDefs.h"

void
InstallMyWriteType(void)
{
	BMimeType mime;

	mime.SetType("text/x-MyWrite");
	mime.SetShortDescription("MyWrite text document");
	mime.SetLongDescription("MyWrite text document");

	// Set the icon for the type using the data stored in a constant
	// data structure defined in our IconDefs.h. This function call
	// is not documented in the Be Book because it was introduced into
	// the API in Haiku.
	mime.SetIcon(kVectorIconBits, sizeof(kVectorIconBits));

	// Set the extension for our type
	BMessage extMsg;
	extMsg.AddString("extensions", "mywr");
	mime.SetFileExtensions(&extMsg);

	// This sets the signature of the preferred application for
	// opening our file type.
	mime.SetPreferredApp("application/x-vnd.test.MyWrite");

	// If we were doing this on BeOS R5 or Zeta, a call to
	// BMimeType::Install() would be necessary to make all of the
	// changes at once. Haiku does not have this requirement and updates
	// the type as each method is called.
}
```

Nothing in the above code should look terribly surprising, especially if you have ever used the FileTypes preferences application. Each of the methods sets basic information about the new type. There is another optional method call which is quite handy: `SetSnifferRule()`. Sniffer rules have nothing to do with airplane glue; instead, they are all about automatic identification of files of your new type. The only problem with MIME sniffer rules is their syntax – it's incredibly flexible, making simple rules merely a little weird-looking and the more complicated ones enough to give you a headache. No matter. The convenience they offer is almost always worth the trouble, which isn't much in the vast majority of cases.

Automagical File Identification: MIME Sniffing

A MIME sniffer rule always follows the following format:

```
rating [begin:end] ( [begin:end] 'pattern1' | ... )
```

At face value, this strange-looking text amounts to a priority rating, a byte range, and a set of patterns placed inside a pair of parentheses. This is only partially true, however. At the lowest level in a rule are what we'll call a **pattern pair**. This consists of a text-matching pattern and an optional byte range for the pattern. The pattern itself must be placed inside a pair of single quotes. One or more pattern pairs can be placed inside a pair of parentheses, separated by a single pipe character (`|`). The example below shows three pattern pairs.

```
([0:15] 'Sample' | 'Another Sample' | [10:] 'Third example')
```

The first pair searches for the word *Sample* in the first 15 characters of the file. The second pair matches only if *Another Sample* shows up at the beginning of the file. The third pair starts searching for *Third example* starting at the tenth byte of the file and continuing to the file's end.

The next level up from a pattern pair in a MIME sniffer rule is the **pattern set**. A pattern set is an optional byte range paired with one or more pattern pairs inside a set of parentheses. The byte range used in a pattern set should be considered a fallback range – it is only used for a pattern pair if the pair does not specify a byte range.

At the top level, a MIME sniffer rule is just a priority rating and one or more pattern sets. The priority is simply a floating point number from 0.0 to 1.0 which rates it in comparison to other rules for the MIME type. Each pattern set is considered to be logically chained together with a boolean AND operator – all pattern sets must match for the rule to match. Below is an example of how this works.

```
.5 ('foo') ([20:] 'BAR') ([0:10] 'baz')
```

This rule has a priority of 0.5 and requires *foo* to be at the beginning of the file, *BAR* to be somewhere in the file after the 19th byte, and *baz* to be somewhere in the first ten bytes.

Beyond all of the confusion that this may cause, there are a few other options in constructing patterns which come in handy. First, `-i` can be placed in front of the first pattern pair in a pattern set to make all of the patterns in the set case-insensitive. Second, a mask can be applied to any pattern to specify certain bytes which do or do not matter by using an ampersand (`&`) in a pattern pair. Third, octal and hexadecimal values can be specified using `\` or `\x` as an escape prefix. Fourth, floating point values can be specified using scientific notation. These options can be observed in the examples below, taken from the private Haiku header file `Parser.h`.

```
200e-3 (-i 'ab')
0.70 ("8BPS  \000\000\000\000" & 0xffffffff0000ffffffff )
```

The first example has a priority of 0.2 and looks for *ab* at the beginning of the file, ignoring case. The second is a little more complicated: in the first ten bytes of the file, the first four bytes are expected to be '8BPS', bytes five and six are ignored, and the next four are expected to be zero. The ignored bytes are specified by the zero bytes in the mask. This second rule is used by Haiku to identify Adobe Photoshop files, which always begin with a series of bytes which look like this:

```
38 42 50 53 00 01 00 00 00 00 00 00
```

The first four of these bytes is the string 8BPS, the next two are a two-byte integer with a value of one, and the last six are reserved bytes which are always supposed to be zero.

Type-Specific Attributes

Some file types have specific attributes associated with them. For example, Ogg Vorbis and MP3 files use the custom attributes `Audio:album`, `Audio:artist`, and `Audio:title` to store information traditionally kept in tags. Should your new type also have a use for extra attributes, it's a pretty simple operation to add them. You will store information about them in

a BMessage and pass the message to `BMimeType::SetAttrInfo()`. Let's say that we wanted to tweak the Person file type to have separate attributes for a person's first and last names. Here is the code that you would write to set these two custom attributes.

```
#include <Message.h>
#include <Mime.h>

int
main(void)
{
	BMimeType mime("application/x-person");
	BMessage attrMsg;

	// We get the existing information for Person files because otherwise
	// we will replace it with the two measly attributes that we have
	// below. We want to add to the existing attributes for Person files,
	// not replace them.
	mime.GetAttrInfo(&attrMsg);

	// Each of these fields needs to be added to the message for the
	// custom attribute to be useful.
	attrMsg.AddString("attr:public_name", "First Name");
	attrMsg.AddString("attr:name", "META:firstname");
	attrMsg.AddInt32("attr:type", B_STRING_TYPE);
	attrMsg.AddBool("attr:viewable", true);
	attrMsg.AddBool("attr:editable", true);

	// These three fields are not documented in the Be Book, but are used
	// by Tracker to determine how the information is displayed in a
	// Tracker window.
	attrMsg.AddInt32("attr:width", 120);
	attrMsg.AddInt32("attr:alignment", B_ALIGN_LEFT);
	attrMsg.AddBool("attr:extra", false);

	attrMsg.AddString("attr:public_name", "Last Name");
	attrMsg.AddString("attr:name", "META:lastname");
	attrMsg.AddInt32("attr:type", B_STRING_TYPE);
	attrMsg.AddBool("attr:viewable", true);
	attrMsg.AddBool("attr:editable", true);
	attrMsg.AddInt32("attr:width", 120);
	attrMsg.AddInt32("attr:alignment", B_ALIGN_LEFT);
	attrMsg.AddBool("attr:extra", false);

	mime.SetAttrInfo(&attrMsg);
}
```

Final Thoughts

Our new type is ready to use now! To set a file to our new type, we have a couple of options:

1. `BNode::WriteAttr()`
2. `BNode::WriteAttrString()`
3. `BNodeInfo::SetType()`

All three methods work properly on BeOS R5 and Zeta, but as of this writing only the third way works properly under Haiku due to an unresolved bug in the operating system. Your new

file type will show up in the FileTypes preferences application and will look and act just like any other standard system file type.