Programming with Haiku

Lesson 12

Written by DarkWyrm

Attributes

The Be filesystem that Haiku uses allows files to have extended attributes. These attributes are **metadata** – information about a file that is not part of the file's contents. Although a number of other operating systems feature this in their filesystems, none of them have pervasive use of them like Haiku does.

The flexibility that attributes provide is not necessarily obvious at the start. For example, MP3 files can have different attributes, such as artist, album, and so forth. No special tag-reading code is needed for MP3s processed this way. E-mails in Haiku are stored as individual files, so it is trivial to search, for example, for all e-mail from a person which was received after a certain date. Person files, which are used to store contact information, are actually zero-byte files which have all of their information stored in attributes. Some contact managers, such as Mr. Peeps!, have expanded the number of attributes dedicated to a person file and even made them store regular data.

The main C++ functions used to manipulate attributes are methods provided by the BNode class.

```
status_t GetAttrInfo(const char *name, attr_info *info) const;
```

Gets the type and size of the attribute named `name` and places it into the corresponding properties of `info`, which must be already allocated when the call is made. `B_ENTRY_NOT_FOUND` is returned if the node does not have the specified attribute and `B_FILE_ERROR` is returned if the BNode is uninitialized. `B_OK` is returned on success. Below is the declaration for the `attr_info` structure:

```
typedef struct attr_info
{
      uint32      type;
      off_t       size;
} attr_info;
```

```
ssize_t ReadAttr(const char *name, type_code type, off_t offset,
                     void *buffer, size_t length);
```

`ReadAttr()` reads the data from attribute `name` and copies up to `length` bytes into `buffer`. As of this writing, `offset` is not used. The type parameter can, like BMessage identifier constants, be any 4-byte integer value, but it is normally one of the predefined constants found in TypeConstants.h, such as `B_STRING_TYPE` or `B_INT32_TYPE`. The number of bytes actually read is returned.

```
ssize_t WriteAttr(const char *name, type_code type, off_t offset,
                void *buffer, size_t length);
```

Dealing with `WriteAttr()` is almost exactly the same as with `ReadAttr()` except that the attribute is erased and the data in `buffer` replaces it. **Note: Indexed attributes**,which are those that can be accessed via a query, **may be no bigger than 255 bytes**. They also must be one of the following types: string, `int32`, `uint32`, `int64`, `uint64`, `float`, or `double`.

```
status_t RemoveAttr(const char *name);
```

This method deletes the specified attribute and returns `B_OK` if successful.

```
status_t RenameAttr(const char *oldname, const char *newname);
```

This performs more of a move than a rename. Beware that if there is an existing attribute named `newname`, it will be clobbered.

```
status_t ReadAttrString(const char *name, BString *out) const;
status_t WriteAttrString(const char *name, const BString *data);
```

These undocumented functions are wildly convenient if you work with string attributes. The data in the named string is placed in `out` in the read version and `data` is written to disk in the write version.

Here is an example of some code which reads an attribute. In this example, we read the E-mail Name attribute.

```
#include <fs_attr.h>
#include <Node.h>
#include <String.h>

BString
GetEmailName(const char *path)
{
	BString out;
	BNode node(path);
	if (node.InitCheck() != B_OK)
		return out;

	// This is to make sure that the attribute exists and what
	// its size is.
	attr_info attrInfo;
	if (node.GetAttrInfo("META:name", &attrInfo) != B_OK)
		return out;

	// BString::LockBuffer() and UnlockBuffer() allow us to have
	// direct access to the internal character buffer that the BString
	// uses. LockBuffer() takes one parameter which is the maximum
	// size that the character buffer needs to be.
	char *nameBuffer = out.LockBuffer(attrInfo.size + 1);
	node.ReadAttr("META:name", attrInfo.type, 0, nameBuffer,
			attrInfo.size);
	nameBuffer[attrInfo.size] = '\0';
	out.UnlockBuffer();

	return out;
}
```

For arbitrary attributes, this works quite well. However, there are certain attributes which are system standards for which a class was created to simplify our lives and save us poor, helpless developers from having to remember these most common attribute names. This includes icons and file types. This help comes in the form of the BNodeInfo class.

BNodeInfo

```
status_t GetAppHint(entry_ref *app_ref);
status_t SetAppHint(const entry_ref app_ref);
```

The app hint for a file suggests to the system which application should be used to open this particular file. `app_ref` is considered a hint because it may end up pointing to something that isn't an application or some other problem that would prevent it from opening the file. For those curious, this information is stored in the attribute "BEOS:PPATH". These two methods aren't used very often.

```
status_t GetIcon(BBitmap *icon, icon_size size = B_LARGE_ICON);
status_t SetIcon(const BBitmap *icon, icon_size size = B_LARGE_ICON);

status_t GetIcon(uint8 **data, size_t *size, type_code *type) const;
status_t SetIcon(const uint8 *data, size_t size);

static status_t GetTrackerIcon(entry_ref *ref, BBitmap *icon,
                               icon_size which = B_LARGE_ICON)
```

The first two of these methods act upon the attribute of the file. "BEOS:M:STD_ICON" holds a 16x16 pixel 256-color icon and "BEOS:L:STD_ICON" is used for a 32x32 pixel, 256-color icon. These are standard attributes for all BeOS operating systems. Haiku introduces another one, "BEOS:ICON", which is used to store the vector icon. Unlike the other two, the data for this icon is stored in the Haiku Vector Icon Format (HVIF) format. The versions of these two calls which do not use the BBitmap class were created to work specifically with the vector icon for a file. `GetTrackerIcon()` gets the icon for the file which would be shown by Tracker which may not necessarily be the same icon as what would be returned by `GetIcon()`, which will be better explained in a moment. In most cases, if you want to get the icon for a file, use `GetTrackerIcon()`.

```
status_t GetPreferredApp(char *signature, app_verb = B_OPEN);
status_t SetPreferredApp(char *signature, app_verb = B_OPEN);
```

These two methods deal with the preferred application for a file. This only deals with this specific file and not the file's type in general. The attribute used here is "BEOS:PREF_APP".

```
status_t GetType(char *type);
status_t SetType(const char *type);
```

Set and get the file's type. This is always a MIME string. Note that if this attribute doesn't exist, calling the global function `update_mime_info()` may help. The attribute on which these methods operate is "BEOS:TYPE".

Note that sometimes it is not as convenient to use the BNodeInfo class' methods as it is with those of BNode, working directly with the attributes. Why? Because BFile is a child class of BNode, it is often easier to reuse an existing BFile object – especially using the `ReadAttrString()` and `WriteAttrString()` methods.

Think Locally, Act Globally

One of the ways in which Haiku is customizable for more advanced users is the way in which it handles the file attributes we examined a moment ago. The global settings are stored in the system's MIME database and manipulated with the BMimeType class. However, individual files can be customized to override these settings.

Let's say for a moment that .xyz files are opened with XYZEdit. This information can be seen and changed in the FileTypes preferences application or programmatically, but there is one file called SpecialFile.xyz which we want to always open with XYZOtherEdit. This can be set using the FileType Tracker add-on, which modifies the "BEOS:PREF_APP" attribute on the individual file. When SpecialFile.xyz is double-clicked, Tracker will open it in XYZOtherEdit. All other .xyz files are opened in the regular editor. If you want to make changes for all .xyz files, then using the BMimeType class is necessary. We'll study that class in detail later on.

Closing Thoughts

The existence of attributes in the BFS filesystem is not unique. Other filesystems, such as XFS and ReiserFS have them, as well. What makes Haiku different is that the major operating systems (Linux, Windows, OS X) do not leverage them extensively, which is sad if you consider the many ways that they can be used. If you poke around in Haiku, you'll find them used for many different tasks and in different ways. Next time we'll look at one powerful operating system feature which centers around them: queries.

Going Further

- Using the Terminal commands `listattr` and `catattr`, see what attributes are used by these kinds of files: People file, an application, an e-mail, and an MP3.
- If you were going to create a Task file for a to-do list application, the Haiku way of storing all of a task's details would be using attributes. Using the general naming scheme from an e-mail's attributes ("MAIL:subject", etc.) as a guide, what would be the names and types of some of the attributes you would use for a task file's information?