Programming with Haiku

Lesson 14

Written by DarkWyrm

Monitoring Nodes

Nodes are dumb. Check the *Be Book* – you'll find the exact same thought there, too. They're always lost because they don't know where they are in the filesystem. Truth be told, though, it's because BNode objects are concerned with an entry's data, not metadata like its name and location. They are also incredibly handy because we can ask the operating system to notify us of changes in the filesystem, such as the mounting and unmounting of volumes, the creation of new files, the modification of a file's attributes, or news of other interesting file-related events.

Setting up node monitoring is relatively easy, but first you will need to decide on the kinds of changes for which you wish to receive notifications. There are five types of changes that can be watched.

Mode	Description
`B_WATCH_NAME`	Watch for changes in file names, including moving or deleting nodes.
`B_WATCH_STAT`	Watch a file's features manipulated by the `stat()` function, including creation and modification times, size, permissions, and ownership.
`B_WATCH_ATTR`	Watch the file's attributes, including adding or removing them.
`B_WATCH_DIRECTORY`	This only works for directories. It will watch for the creation, deletion, or renaming of entries in the directory. If used on a file, it won't do anything.
`B_WATCH_ALL`	Watches for everything described above.
`B_WATCH_MOUNT`	Instead of watching a file or directory, using this flag will cause you to be notified of volumes being mounted or unmounted. The `nref` argument isn't needed if this is the only flag used in a `watch_node()` call. `B_WATCH_ALL` doesn't include this flag.
`B_STOP_WATCHING`	Turns off watching on the node pointed to by `nref`.

```
watch_node(const node_ref *nref, uint32 flags, BMessenger messenger);
watch_node(const node_ref *nref, uint32 flags, const BHandler *handler,
            const BLooper *looper = NULL);
```

These calls start the node monitor to watch the node pointed to by `nref` for changes specified in `flags`. Messages are sent by the BMessenger messenger or to the BHandler or BLooper specified. Note that the target of the BMessenger must be within the application calling `watch_node()`. The system has only 4096 monitoring slots, so don't go overboard. Also, each call to `watch_node()` consumes a slot even if the node is already watched by the monitor.

```
status_t stop_watching(BMessenger messenger);
status_t stop_watching(const BHandler *handler,
                       const BLooper *looper = NULL);
```

These functions end monitoring for any nodes for which messages are sent to the specified target, i.e. if all node monitoring messages are sent to the same target, this frees all slots in one fell swoop.

The Node Monitor's Update Messages

As nice as live updates are to have in your application, handling the Node Monitor's update messages can get a bit complicated. The reason for this is a combination of handling the various update messages and mapping them to appropriate actions for your application.

The update messages themselves are strikingly similar to update messages sent by live queries. The `what` field of a node monitor message is `B_NODE_MONITOR`. Checking the `opcode` field of this message gives you more information about what kind of update you're receiving. The opcode is an `int32.`

Opcode B_ENTRY_CREATED

Conditions for receiving:

- `B_WATCH_DIRECTORY` on directory in which the node was created

Field	Type	Description
`name`	string	Name of the new entry.
`directory`	int64	The node number (`ino_t`) for the new entry's parent directory.
`device`	int32	The id (`dev_t`) of the volume on which the entry was created.
`node`	int64	The node number of the new entry.

If you watch a directory, this is one possible message you'll receive. Of all of the update messages, this one is the most helpful. The `name`, `device`, and `directory` fields can be used to construct an `entry_ref` which points to the new entry. `device` and `node` can be used to create a `node_ref`, useful if you want to monitor the new entry's node. `device` and `directory` can be used to also create a `node_ref`, but one which points to the entry's parent directory and is suitable for initializing a BDirectory object. If you plan on handling `B_ENTRY_REMOVED` opcodes, you'd better make both an `entry_ref` and a `node_ref` for the new entry and stash it away somewhere for later use.

Opcode B_ENTRY_REMOVED

Conditions for receiving:

- `B_WATCH_DIRECTORY` on directory in which the node resided
- `B_WATCH_NAME` on the former node

Field	Type	Description
`directory`	int64	The node number (`ino_t`) for the former entry's parent directory.

Field	Type	Description
device	int32	The id (dev_t) of the volume from which the entry was removed.
node	int64	The node number of the former entry. Of course, because the node no longer exists, it is invalid and useless except for comparing against cached values.

Notifications for removed entries require extra work to be useful, sadly. The problem lies in the fact that there is no name field sent, preventing construction of an entry_ref, which just happens to be the most common way of storing an entry's location without consuming a file descriptor. As a result, if you plan on handling removal notifications, you will need to save away both a node_ref and an entry_ref for each entry on the filesystem which you wish to track. There will be more on this later.

Opcode B_ENTRY_MOVED

Conditions for receiving:

- B_WATCH_DIRECTORY on the source or destination directory
- B_WATCH_NAME on the node itself

Field	Type	Description
name	string	Name of the moved entry.
from directory	int64	The node number (ino_t) for the entry's source directory.
to directory	int64	The node number (ino_t) for the entry's destination directory.
device	int32	The id (dev_t) of the volume on which the entry resides.
node	int64	The node number of the moved entry. This doesn't change even though the entry has moved.

Handling B_ENTRY_MOVED is almost the same as B_ENTRY_CREATED except for the changes in directory field names.

Opcode B_STAT_CHANGED

Conditions for receiving:

- B_WATCH_STAT on the node itself

Field	Type	Description
device	int32	The id (dev_t) of the volume on which the entry resides.
node	int64	The node number of the entry.

Opcode B_ATTR_CHANGED

Conditions for receiving:

- `B_WATCH_ATTR` on the node itself

Field	Type	Description
`device`	int32	The id (`dev_t`) of the volume on which the entry resides.
`node`	int64	The node number of the entry.

> Changes in `stat` data are as big of a pain to handle as entry removals and for the same reason: having to stash away `node_refs`. The Be Book even recommends storing away a copy of each entry's `stat` data, too, which wouldn't be a bad idea as long as you're careful about memory usage. Handling attribute changes are pretty much the same thing.

Opcode B_DEVICE_MOUNTED

Conditions for receiving:

- `B_WATCH_MOUNT` flag used with `watch_node()`.

Field	Type	Description
`device`	int32	The id (dev_t) of the volume on which the new volume's mount point resides
`new device`	int32	The id (dev_t) of the new volume.
`node`	int64	The node number of the entry.

Opcode B_DEVICE_UNMOUNTED

Conditions for receiving:

- `B_WATCH_MOUNT` flag used with `watch_node()`.

`device`	int32	The id (dev_t) of the former volume.

> Volume-related notifications don't require very much effort, thankfully. Do be aware that `dev_t` numbers are apparently recycled fairly often, but the only time you'll ever need it is if you've been tracking volumes and keeping a copy of each volume's `dev_t` identifier stored away somewhere.

Handling Update Messages

At this point you may be wondering how all of these messages fit together or how to use them in your own programs. The work itself isn't very difficult, but it can be tedious. The way node monitoring fits into your program largely depends on what kind of program you're writing and the purpose for which you want notifications. If you are working on a file manager of some kind, you'll probably be monitoring everything, including volume-related events. This is definitely the more complicated scenario. More likely, though, you'll just be monitoring the node for a document your program is editing, in which case your job is

relatively easy. We'll be looking at the simpler of the two cases since the file manager scenario is very implementation-specific and is just an extension of the other.

Let's say for this instance that we are writing a simple text editor. Architecturally, we'll say that each actual document is encapsulated by a Document class and BView subclass has been created for the editor, predictably called DocumentEditor. What we wish to accomplish with node monitoring is to be informed of outside changes to the document currently being edited.

The changes we will need to handle in this case are removal, moving, and `stat` changes. It's possible to merely ignore removals, but we want to ensure that the user's data is kept safe – if the user didn't have any changes made and closed the window, the contents of the document stored on the disk and in memory would be lost, potentially leading to frustration and gnashing of teeth. It would be better to ask if the user wishes to re-save the document.

Setup Before Watching Files

The removal and `stat` notifications received from the node monitor only have enough information for creating `node_ref` structures, so our document class will need to store a `stat` structure and a `node_ref`. The Document class could look something like this:

```
#include <Node.h>
#include <sys/stat.h>

class Document
{
public:
                    Document(const char *path);
                    ~Document(void);

        const char *GetName(void) const;

        status_t    Load(const char *path);
        void        NodeMoved(const entry_ref &ref);

        node_ref    GetNodeRef(void) const;
        entry_ref   GetRef(void) const;
        struct stat GetStatData(void) const;

        // various methods here

private:
        entry_ref   fRef;
        node_ref    fNodeRef;
        struct stat fStatData;

        // More document-related properties here
};
```

Keeping an `entry_ref` around is a cheap way of storing the location of the document without using a file descriptor. `fNodeRef` and `fStatData` are kept around for node monitoring purposes. The `Load()` method will need to set these properties if everything else about the document has been successful. `NodeMoved()` is necessary because the owning DocumentEditor instance will receive the notifications from the Node Monitor and will need to pass on information so that in the event that the user – or something else, for that matter – moves the file while it is open, the editor doesn't try to save the document in the old location.

The `Load()` method could look something like this:

```
Document::Load(const char *path)
{
	BFile file(path, B_READ_ONLY);
	if (file.InitCheck() != B_OK)
		return file.InitCheck();

	// A bunch of data loading muck goes here. Nothing to see here.
	// Move along, now. ;-)

	// Assuming that everything went well in loading the document,
	// let's save the info about the file on the disk.
	BEntry entry(path);
	entry.GetRef(&fRef);
	entry.GetNodeRef(&fNodeRef);
	entry.GetStat(&fStatData);
}
```

There! Now that the initial setup has been taken care of, we can move on to the DocumentEditor class. We'll just say that it has a `Load()` method, too.

```
#include <NodeMonitor.h>

status_t
DocumentEditor:::Load(const char *path)
{
	status_t status = fDocument.Load(path);
	if (status != B_OK)
		return status;

	// Document loaded OK, so set up node monitoring. BView inherits
	// from BHandler, so this is an easy call.
	node_ref nref = fDocument.GetNodeRef();
	watch_node(&nref, B_WATCH_NAME | B_WATCH_STAT, this);
}
```

Fielding Update Messages

With node monitoring now set up, all that is left is to tweak `MessageReceived()` to handle the update messages.

```
DocumentEditor::MessageReceived(BMessage *msg)
{
	switch (msg->what)
	{
		// A bunch of other message-handling cases here

		case B_NODE_MONITOR:
		{
			// We'll use a separate function to prevent making
			// MessageReceived() any messier than it already tends
			// to be.
			HandleNodeMonitoring(msg);
			break;
		}
		default:
```

```
		{
			BView::MessageReceived(msg);
			break;
		}
	}
}


void
DocumentEditor::HandleNodeMonitoring(BMessage *msg)
{
	int32 opcode;
	if (msg->FindInt32("opcode", &opcode) != B_OK)
		return;

	switch (opcode)
	{
		case B_ENTRY_REMOVED:
		{
			// Document has been lost on disk. Alert user.
			BString errmsg;
			errmsg += fDocument.GetName();
			errmsg << " has been deleted on the disk. Do you wish "
				<< "to re-save it to prevent data loss?";
			BAlert *alert = new BAlert("MyCoolEditor",
									errmsg.String(),
									"No", "Yes");
			if (alert->Go() == 1)
				fDocument.Save();
			break;
		}
		case B_ENTRY_MOVED:
		{
			dev_t device;
			ino_t destDir;
			BString name;
			if (msg->FindInt64("to directory", &destDir) == B_OK &&
				msg->FindInt32("device", &device) == B_OK) &&
				msg->FindString("name", &name) == B_OK)
			{
				entry_ref newRef;
				newRef.setname(name.String());
				newRef.device = device;
				newRef.dir = destDir;
				fDocument.NodeMoved(ref);
			}
			break;
		}
		case B_STAT_CHANGED:
		{
			// Check to see what has changed. We'll ignore (for now)
			// changes in permissions, but a change in
			// modification time means the file has changed, in which
			// case we'll ask about reloading from disk.
			struct stat oldStat = fDocument.GetStatData();

			struct stat newStat;
			entry_ref ref = fDocument.GetRef();
			BFile file(&ref);
```

```
                        file.GetStat(&newStat);
                        if (newStat.st_mtime != oldStat.st_mtime)
                        {
                              BString errmsg;
                              errmsg += fDocument.GetName();
                              errmsg << " has been changed on the disk. Do you "
                                    << "wish to reload it??";
                              BAlert *alert = new BAlert("MyCoolEditor",
                                                errmsg.String(),
                                                "No", "Yes");
                              if (alert->Go() == 1)
                              {
                                    BMessage refMsg(B_REFS_RECEIVED);
                                    refMsg.AddRef("refs", fDocument.GetRef());
                                    Window()->PostMessage(refMsg);
                              }
                        }

                        break;
                  }
                  default:
                        break;
            }
}
```

In case you're wondering, `Document::NodeMoved()` just uses the same BEntry code as in `Load()` to update the document's `node_ref` and `stat` data.

Going Further

- Give some thought to how node monitoring might be used in a file manager application.
- Look over the code from the open source file manager Seeker to see how node monitoring was used to track entries in the currently-open folder.