Proceedings of FREENIX Track:
2000 USENIX Annual Technical Conference

San Diego, California, USA, June 18–23, 2000

UNIX FILE SYSTEM EXTENSIONS IN THE GNOME ENVIRONMENT

Ettore Perazzoli

Unix file system extensions in the GNOME environment

Ettore Perazzoli
Helix Code, Inc.
ettore@helixcode.com, http://primates.helixcode.com/~ettore/

1 Introduction

This paper explains how the GNOME [GNOME] project is extending the functionality of the Unix file system for use in both the desktop and applications, by using a user-level library called the GNOME Virtual File System.

There are various reasons for extending the Unix file system and its API, as explained in the following sections.

1.1 Uniform file access

In a modern desktop environment, users have to deal with files that are available in different ways. For example, files can be remotely available from a Web or FTP site. Sometimes they are stored locally, but are contained in tar or zip files. Sometimes they are stored in a compressed form.

Each of these cases usually requires the user to use a different tool; using different tools means that users have to learn the user interface for each of them, and have to manually make these tools talk to the applications, for example by using temporary files.

To avoid this problem, there should be a global file system namespace that all applications can use to access these different kinds of files. Users should not need to install and use different programs to access files available through different methods, and all the applications should be able to access this global file system namespace.

So basically we need a consistent API that all the applications can use to access these files in a simple way.

1.2 Representing special non-file objects

Current Unix desktop environments lack a unified file system -like representation of the system resources. In the Windows operating system, users can access all the system resources from the "My Computer" folder which acts as the root of the operating system shell's "namespace". This means that the control panel, the printer's spool, the trashcan, the removeable devices and so on are all accessible through the same simple "point-and-click" mechanism within the same application (`Explorer.exe`). Unlike the current Unix desktop environments, users don't need to deal with different applications for the various objects that are in their computer: they can just use the desktop shell.

To implement similiar functionality on Unix, we would need to have an extensible mechanism for providing the contents to these virtual folders and implementing operations that can be performed on them.

1.3 Asynchronous operation

On Unix, there is no API to let applications do fully asynchronous I/O in an easy and portable way.

The reason why this is so important is that GUI applications need to be responsive all the time. If a GUI application is blocked during a long synchronous file operation, the user cannot stop the operation, nor perform any other task with the same application. This is especially important for a file manager, which needs to be able to perform multiple I/O tasks in parallel without taking control away from the user.

One of the ways of dealing with asynchronous I/O, i.e. using the select() system call, does not work with all the operations; for example, there is no way to make an asynchronous `gethostbyname()`, `open()` or `stat()` without complicated hacks.

Unix programmers could also use POSIX threads for performing asynchronous operations, but thread programming is very error-prone and can easily lead to problems. Writing a threaded application requires more programming skills writing a non-threaded one, and in the free software world it is very important to make things easy for programmers.

Moreover, POSIX threads are not fully portable across different platforms, as not all implementations are reliable and efficient.

Finally, although a POSIX API for asynchronous I/O exist, it is not implemented on all systems and does not work with all system calls. For example, you cannot execute the `open()` nsystem call in an asynchronous fashion. Moreover, this API does not fit nicely in the event-based model of GUI systems; consequently, it is not suitable for rapid development of GUI applications.

What we need for asynchronous operation is an easy to use API that hides most of the details from the programmer and integrates nicely with the existing GUI toolkits. The API should give a better abstraction for doing async I/O, in the most simple way possible.

2 Existing User Space Virtual File Systems

In the past, there have been other attempts to implement extensions to the Unix file system in the free software world, none of which completely fulfilled the needs of the GNOME project.

2.1 The GNU Midnight Commander

The GNU Midnight Commander is the file manager of choice for the GNOME project at the time of writing. It implements a user-space Virtual File System library that solves the problem of accessing files within archive files or on remote Internet sites through an extended URI system which is explained in greater detail in section 4.

Although the VFS library source code could be made independent of the GNU Midnight Commander and thus its functionality could be used by all the applications, it suffers a few design problems:

- It does not support asynchronous operation: if you perform an operation on a remote system, the Midnight Commander will be blocked until the operation is complete, and the user is not even able to stop the operation.
- The library is hidden under a Unix API that is not very extensible.

The latter problem is the most important one: as a Virtual File System has to deal with kinds of files that can be quite different from the standard Unix files, it needs some API extensions to support this functionality. Moreover, it needs some API extensions to deal with asynchronous operations. Unfortunately, it is not possible to add functionality to the API in a clean way without breaking Unix compat- ibility, so this makes the GNU Midnight Commander's Virtual File System unsuitable as a generic library for GNOME.

2.2 KDE's `kio`

The K Desktop Environment [KDE] provides a completely asynchronous virtual file system library called `kio`.

`kio` uses the same extended URI syntax that the GNU Midnight Commander uses; but unlike the Midnight Commander, all the file operations are performed through external helper processes (`kioslaves`) in an asynchronous fashion. The helper process communicates with the master process using Unix domain sockets and custom protocols; the library performs encoding/decoding of such protocols automatically and notifies the programmer using the Qt [Qt] signal system.

Although this system is closer to the requirements that we have listed at the beginning of this paper, it is still suboptimal, for the following reasons:

- Operations are very simple: the API allows the programmer to download/upload a whole file and perform some simple file operations, but it does not provide all the versatility of the Unix API.
- Each file operation requires an external process: `kio` does not take advantage of threads.
- There is no API to perform operations in a synchronous fashion, so the programmer needs to spawn a process, request an operation and then wait for the result from it.

3 The GNOME Virtual File System

The solution we gave to these issues in the GNOME project is in the form of a new library, designed from scratch to meet all the requirements. This library (the GNOME Virtual File System, or GNOME VFS for short) takes advantage of the existing GNOME development libraries, such as GLIB and GTK+ [GTK].

The following design ideas were kept in mind while implementing the GNOME VFS:

- Like the rest of GNOME, the API should be C-based and follow the standard GNOME programming style.
- The implementation should be portable, so that making GNOME VFS a core component of GNOME would not restrict the availabilty of GNOME on various Unix platforms.
- Using GNOME VFS should not make the programmer's life harder.
- The asynchronous API should be simple to use, and be nicely integrated with the standard GLIB main loop that is central to GNOME applications. (The GLIB main loop is the main event-handling loop in GTK+ and GNOME applications. The X main loop is nicely wrapped by the GLIB loop.)
- Adding new access methods should be possible, and programmers should not need to care about too many of the details, such as asynchronous behavior. (By "access method", we mean the implementation of a protocol, of a file format, or any other way to retrieve, create or modify a GNOME VFS file. For example, there should be an HTTP access method, a ZIP file access method and so on.)

4 Extended URIs

The GNOME VFS uses an extension of the traditional web Uniform Resource Identifiers (URIs) to access files, instead of the standard Unix file paths. This enables us to access different kinds of resources in a way that is both generic and easy to understand for users.

GNOME VFS extended URIs are reminiscent of the GNU Midnight Commander's URI syntax, and consequently use the `#` character to stack access methods on top of each other.

In the simplest form, a GNOME VFS URI looks exactly like a normal Web URI, that is:

```
method://user:password
@some.host/path/to/file
```

As in normal Web URIs, user and password are optional; in that case, the `@` character will be omitted as well.

For example, the URI

```
http://www.gnome.org/index.html
```

will refer to the file `index.html` from the host `www.gnome.org` through the `http` access method, which is an implementation of the HTTP protocol.

The access method for the local file system is called `file`, so, for example, the file `/etc/passwd` is accessed through the URI

```
file:/etc/passwd
```

Unlike normal Web URIs, though, GNOME VFS URIs let you "stack" access methods on top of each other. GNOME VFS, in fact, supports two kinds of access methods: "toplevel" access methods, that access files directly, and "archive" access methods, that access files within other files.

For example, there is a `zip` access method that lets you access files contained in a `.zip` archive file: the `.zip` access needs a "parent" access method to access the archive in which the file is contained.

Stacking is achieved in GNOME VFS URIs by using the special character `#`. The generic syntax is:

```
uri#method[/sub_uri]
```

When this syntax is used, it specifies that `sub_uri` must be accessed within the file available through uri using the specified method.

For example, imagine you have a `foo.zip` archive containing a file called `bar.c`. Also suppose that `bar.c` is contained in a directory called `baz` within `foo.zip`, and that `foo.zip` is in your home directory `/home/joe`. The URI to specify `foo.zip` is:

```
file:/home/joe/foo.zip\#zip/baz/bar.c
```

If the file was available through FTP instead, the URI would be something like:

```
ftp://joe:passwd@ftp.site.net
/home/joe/foo.zip#zip/baz/bar.c
```

Some access methods don't require a subpath; for example, this is the case with the `gzip` access method that can be used to read and create compressed files in `.gzip` format.

If you wanted to read the contents of a compressed `foo.gz` file in your home directory, you would simply have to specify the following URI:

```
file:/home/joe/foo.gz{\tt \#}gzip
```

By combining the `gzip` access method with the `tar` access method, it is possible to access files contained in tar.gz archives:

```
file:/home/ettore/download
/gnome-vfs-0.1.tar.gz#gzip
#tar/gnome-vfs-0.1/AUTHORS
```

There is no limitation in the amount of stacking that can be performed. Every access method in GNOME VFS is implemented as a dynamically loaded plug-in, as described in section 10: anyone can extend the VFS with new access modules.

5 GNOME VFS API basics

Almost all the standard Unix file operations have conterparts in GNOME VFS. In the following sections, we will give a brief overview of the GNOME VFS API.

5.1 The `GnomeVFSURI` object

A GNOME VFS URI is represented by a special GNOME VFS object called `GnomeVFSURI`. `GnomeVFSURI` objects are the preferred way to specify files: users of the library can use `GnomeVFSURI`s to store and manipulate URIs, and the interface between the library and its plug-ins uses `GnomeVFSURI` objects.

A `GnomeVFSURI` object is created by using the call

```
GnomeVFSURI *
gnome_vfs_uri_new (const char *s)
```

`GnomeVFSURI`s also have a reference count that can be controlled by using the calls:

```
void gnome_vfs_uri_ref (GnomeVFSURI *uri)
```

```
void gnome_vfs_uri_unref (GnomeVFSURI *uri)
```

A `GnomeVFSURI` can be also converted into a printable string by using the following call:

```
char *gnome_vfs_uri_to_string
    (const GnomeVFSURI *uri,
     GnomeVFSURIHideOptions hopt)
```

It is also possible to extract the host, user name and password information from it, as well as compare and combine them. Virtually any path operation that the programmer might need is supported directly through the GnomeVFSURI API.

5.2 The `GnomeVFSResult` enumeration

All the GNOME VFS operations return a result value of type `GnomeVFSResult` that represents the result of the operation. `GnomeVFSURI` is a numeric enumeration:

```
enum _GnomeVFSResult {
    GNOME_VFS_OK,
    GNOME_VFS_ERROR_NOTFOUND,
    GNOME_VFS_ERROR_GENERIC,
    GNOME_VFS_ERROR_INTERNAL,
    GNOME_VFS_ERROR_BADPARAMS,
    GNOME_VFS_ERROR_NOTSUPPORTED,
    GNOME_VFS_ERROR_IO,
    GNOME_VFS_ERROR_CORRUPTEDDATA,
    /* ... */
    GNOME_VFS_NUM_ERRORS
};
typedef enum _GnomeVFSResult
  GnomeVFSResult;
```

Just like the Unix `errno` variable, you can get a string description from a GnomeVFSURI value by using the following function:

```
const gchar *gnome_vfs_result_to_string
  (GnomeVFSResult result)
```

6 Synchronous API

In GNOME VFS, both a synchronous and asynchronous API call exist most file operations. The synchronous versions work like normal Unix calls: they perform the operation, then return and report success/failure using a GnomeVFSResult value.

6.1 The `GnomeVFSHandle` object

As in the Unix API, files need to be "open" before being ready to be read or written. But while the Unix API returns a simple integer to represent a "file handle", the GNOME VFS API provides a special object type for that, called `GnomeVFSHandle`.

GnomeVFSHandle objects are created using the "open" or "create" calls

```
GnomeVFSResult
gnome_vfs_open_uri
  (GnomeVFSHandle **handle return,
   GnomeVFSURI *uri,
   GnomeVFSOpenMode open mode)
```

```
GnomeVFSResult
gnome_vfs_create_uri
  (GnomeVFSHandle **handle return,
   GnomeVFSURI *uri,
   GnomeVFSOpenMode open mode,
   gboolean exclusive,
   guint perm)
```

and destroyed by using the "close" call:

```
  GnomeVFSResult
  gnome_vfs_close (GnomeVFSHandle *handle)
```

6.2 Synchronous I/O Example

Here is a simple example demonstrating synchronous operation in GNOME VFS. This subroutine will read a file and output its contents to stdout. The code should be rather self-explanatory.

```
gboolean vfs_cat (const char *uri)
{
  GnomeVFSURI *vfs_uri;
  GnomeVFSHandle *handle;
  GnomeVFSResult result;

  vfs_uri = gnome_vfs_uri_new (uri);
  if (vfs_uri == NULL) {
    printf ("‘%s’ is not a valid URI.\n",
      uri);
    return FALSE;
  }

  result = gnome_vfs_open_uri
    (&handle, vfs_uri, GNOME_VFS_OPEN_READ);

  if (result != GNOME_VFS_OK) {
    printf ("Error opening ‘%s’: %s\n",
      uri,
      gnome_vfs_result_to_string (result));
    return FALSE;
  }

  while (1) {
    GnomeVFSFileSize bytes_read;
    GnomeVFSFileSize i;
    char buffer[4096];

    result = gnome_vfs_read
      (handle,
       buffer,
       sizeof (buffer),
       &bytes_read);
    if (result != GNOME_VFS_OK) {
      printf ("%s: %s\n", uri,
        gnome_vfs_result_to_string
          (result));
      return FALSE;
    }

    if (bytes_read == 0)
      break;
    for (i = 0; i < bytes_read; i++)
      putchar (buffer[i]);
  }

  gnome_vfs_close (handle);
  gnome_vfs_uri_unref (vfs_uri);
  return TRUE;
}
```

7 Asynchronous API

In the asynchronous API calls, instead, the operation requested is started in a separate thread of execution and control returns to the caller immediately. The caller will have to specify a callback function that will be called when the operation is completed. If the operation is long to perform (such as a “copy” operation), the callback might be called more than once to report progress.

All the syncrhonous API calls have asynchronous counterparts; their name is the same as the synchronous one, but use the gnome vfs async prefix, instead of just gnome vfs.

Callbacks for asynchronous operations are triggered in the normal GLib/GTK+ event loop. This means that the application will be able to handle GUI events and GNOME VFS events simultaneously and transparently. The programmer just needs to set up the callback functions and make sure the event loop is running all the time. This makes usage of GNOME VFS in GUI applications very convenient and easy to use.

7.1 The `GnomeVFSAsyncHandle` object

If an asynchronous operation is started successfully, the caller is given back a GnomeVFSAsyncHandle object that can be used to stop the operation after it has been started, by using the following call:

```
GnomeVFSResult gnome_vfs_async_cancel
 (GnomeVFSAsyncHandle *handle)
```

In the case of file “open” and “create” operations, the `GnomeVFSAsyncHandle` object can also be used to request read/write operations on the file after it has been opened or created.

7.2 Opening a file as a GLib I/O channel

A common application case is when the program needs to get data from a file stream as soon as it becomes available from the transport layer. In the case of Unix, this is achieved through the select() system call.

GLib abstracts this mechanism by merging it with the event handling loop, through objects known as `GIOChannels`. With `GIOChannel`s, it is possible to attach callbacks to a file descriptor, and have functions called as soon as data becomes available on it.

Special GNOME VFS API functions allow the programmer to open and read (or write) a file through a `GIOChannel`. For example, a file can be opened by using the following function:

```
GnomeVFSResult
gnome_vfs_open_as_channel
  (GnomeVFSAsyncHandle **handle return,
   const gchar *text uri,
   GnomeVFSOpenMode open_mode,
   guint advised_block_size,
   GnomeVFSAsyncOpenAsChannelCallback
     callback,
   gpointer closure);
```

Notice that, unlike the normal Unix `select()` call, this is reliable with local files too: the application will never be blocked after reading from a file for which the "data available" callback has been called.

This is possible because of the GNOME VFS asynchronous engine described in section 11.

7.3 Asynchronous API example

The following function reads a file asyncrhonously, with the output sent to stdout.

```
#define BUFFER_SIZE 4096

/* This is the callback that
   will be called whenever
   something happens on the
   I/O channel associated
   with the file. */

static gboolean
io_channel_callback
 (GIOChannel *source,
  GIOCondition condition,
  gpointer data)
{
  char buffer[BUFFER_SIZE + 1];
  unsigned int bytes_read;
  unsigned int i;

  if (condition & G_IO_IN) {
    /* Data is available. */
    g_io_channel_read
      (source, buffer,
       sizeof (buffer),
       &bytes_read);

    for (i = 0; i < bytes_read; i++)
      putchar (buffer[i]);

    fflush (stdout);
  }

  /* An error happened while reading
     the file. */

  if (condition & G_IO_NVAL)
    return FALSE;

  /* We have reached the end of the
     file. */

  if (condition & G_IO_HUP) {
    g_io_channel_close (source);
    return FALSE;
  }

  /* Returning TRUE will make sure
     the callback remains associated
     to the channel. */

  return TRUE;
}

static void
open_callback (GnomeVFSAsyncHandle *handle,
               GIOChannel *channel,
               GnomeVFSResult result,
               gpointer data)
{
  if (result != GNOME_VFS_OK) {
    printf ("Error: %s.\n",
            gnome_vfs_result_to_string
              (result));
    return;
  }
  printf ("Open: `%s'.\n",
    (char *) data);
  g_io_add_watch_full
    (channel, G_PRIORITY_HIGH,
     G_IO_IN | G_IO_NVAL | G_IO_HUP,
     io_channel_callback,
     handle, NULL);
}
```

```
void
start_cat (const char *uri)
{
  GnomeVFSAsyncHandle *handle;

  /* Start the ‘read’ operation. */
  gnome_vfs_async_open_as_channel
    (&handle, uri, GNOME_VFS_OPEN_READ,
     BUFFER_SIZE, open_callback, "data");
}
```

8 File attributes

GNOME VFS also provides a `GnomeVFSFileInfo` object that works as an extension of `struct stat` in Unix. In addition to the standard attributes that `struct stat` provides, `GnomeVFSFileInfo` also gives access to:

- MIME type information. GNOME VFS is able to take advantage of plug-ins for which the MIME type is part of the informatin that the access method provides. For example, in the case of the HTTP back-end, GNOME VFS can provide the MIME type as returned by the HTTP server.
- Metadata. Arbitrary value/key pairs can be associated with files, for generic purposes. For example, you can use an "icon" value for specifying a file's icon, or a "description" value for a verbose description of the file.

As some kind of information might not be supported by certain plug-ins (for example, it is not possible to know the number of physical blocks occupied by a file via HTTP), `GnomeVFSFileInfo` provides a bitmask specifying which fields are actually valid and which are not.

GNOME VFS also provides a simple API for loading directory information in a progressive way, calling an asynchronous callback as data is copied into memory. This functionality is particularly useful for a file manager, as the directory view can be updated without blocking the user interface and thus giving the user effective visual feedback of what is going on, even with those slow back-ends for which reading a directory is an expensive operation (such as a `tar` file access method, that requires a sequential scan of the whole `.tar` file).

9 File transfer support

In GNOME VFS there is a special API call for copying and moving files, while providing the caller with progress information while the operation is being performed. GNOME VFS is able to automatically optimize the case in which a file is moved through two different locations on the same physical file system, by using the information provided by the source and destination plug-ins.

When this API call is used, all the file transfer is performed in a separate thread or process, which dispatches the progress information to the main thread/process using the same mechanism that is used by the other asynchronous calls.

In order to reduce the impact of dispatching progress information across process/thread boundaries, the actual calls take place periodically, at a rate specified by the programmer. In the case of a file manager, for example, this will be done only a few times in a second, just enough to make sure the GUI is updated the way the user would expect.

10 Implementation of access plug-ins

As explained in 4, access methods are implemented as dynamically loaded plug-ins. Even the file plug-in that accesses local files is implemented as a plug-in.

Dynamic loading of plug-ins happens during GnomeVFSURI parsing (see section 5.1): the URI is split into its #-separated subparts, the library looks up the access method names (such as file, http and so on), and tries to locate the corresponding plug-in library, by using both a system-wide and a user-wide configuration file called gnome-vfs.conf. If the library is located, it gets linked in dynamically; otherwise, a return code is returned, reporting that the URI is invalid.

Every access method module must provide an initialization function; the initialization function returns a simple vtable containing pointers to the implementations of the GNOME VFS operations for that access method. Pointers to these vtables are stored directly into the GnomeVFSURI object, ready for the first operation to be invoked on it.

At the time of writing, the following modules have been implemented:

- A `bzip2` module, for files that are compressed with the bzip program.
- A `gzip` module, for files that are compressed with the `gzip` program.
- An `ftp` method for accessing remote sites through the Internet FTP protocol.
- An `http` method for accessing remote sites through the HTTP 1.1 protocol. This module also supports WebDAV.
- An `extfs` module, supporting Midnight Commander's generic archive file support based on simple shell scripts. This multipurpose module makes GNOME VFS able to deal with `zip`, `zoo`, `rpm`, `deb`, `arj` and other formats.
- A `gconf` protocol to access the new GNOME configuration database, called `GConf`.

Other modules, including a `tar` one, are being developed.

While this is not supported at the time of writing yet, there are also plans to support CORBA-based access method plug-ins, possibly using the Bonobo component model. [CORBA]

This will have important consequences:

- It will become possible to write plug-ins in any CORBA-aware language. For example, you could have plug-ins written in Perl, Python, or other scripting languages for which CORBA support exists. This will make it extremely simple for people to come up with their own special scripts for special directories or file systems.
- It will become possible to make GNOME VFS see file systems that are implemented in a different process. The external process might be running all the time, and would be contacted through CORBA. For example, you see the list of active print jobs in the print spooler as a normal GNOME VFS directory.

11 Implementation of asynchronous operations

To make operations totally asynchronous, we need to be able to perform them independently of the code that wants them to be executed. This can be done using either external helper processes or helper threads.

The thread-based solution has a number of advantages:

- It requires less memory to work.
- It is faster, as no data needs to be transferred from the helper to the master.

Unfortunately, it is not fully portable, as many systems don't have a suitable thread support. For this reason, GNOME VFS implements asynchronous operation in both ways. This is done by splitting the library in two parts: the basic file access library, and the asynchronous wrapper library. While there is one single version of the former, there are two versions of the latter: one that is based on POSIX threads, and one that uses external processes.

At run time, GNOME VFS applications are dynamically linked to either of the two wrapper libraries. This makes it possible to use either method without changes in the source code. In the case of helper processes, the GNOME Virtual File System uses CORBA to communicate with them.

Using CORBA has the advantage of not requiring the creation of a custom inter-process communication protocol; moreover, adding new operations is very simple (you just need to extend the IDL). GNOME comes with its own ORB by default, so this does not add any further constraints to the applications willing to use GNOME VFS.

12 GNOME VFS and GNOME

GNOME VFS is a core component of the upcoming GNOME 2.0 platform. When GNOME 2.0 is released, GNOME VFS will be available for all applications (GUI or non-GUI) to use. GNOME VFS is also a central component of the new GNOME file manager and shell, which is called Nautilus [Nautilus] and is currently being developed by Eazel, Inc. [Eazel]. Nautilus makes extensive use of asynchronous operations and file system abstractions to improve the usability of the GNOME desktop. The first public release of Nautilus is expected in Summer 2000.

13 Availability

GNOME VFS is available from the GNOME FTP repository:

```
ftp://ftp.gnome.org
/pub/GNOME/unstable/sources/gnome-vfs
```

It is also possible to access the source code through the GNOME anonymous CVS system `anoncvs.gnome.org`, as explained at the URL

```
http://developer.gnome.org/tools/cvs.html
```

The name of the module is `gnome-vfs`.

The GNOME CVS has a web front-end available at the URL

```
http://cvs.gnome.org
```

References

[GNOME] The GNOME home page, `http://www.gnome.org`

[HelixCode] Helix Code, Inc. `http://www.helixcode.com`

[KDE] The KDE home page, `http://www.kde.org`

[Qt] Qt `http://www.troll.no`

[GTK] GTK+ `http://www.gtk.org`

[Nautilus] Nautilus `http://nautilus.eazel.com`

[CORBA] CORBA `http://www.corba.org`

[Eazel] Eazel, Inc. `http://www.eazel.com`