Linux Journal Issue #123/July 2004

Features

Rapid Application Development with Python and Glade *by David Reed*

When you're writing complicated business apps, spend more time on your business logic and let GladeGen do the rest.

Cross-Platform Network Applications with Mono *by Ian Pointer*

Build and run a useful blogging app and get a jump on .NET-compatible development.

Developing for Windows on Linux *by Joey Bernard*

Use these tools from the MinGW Project to write, maintain and test Win32 apps on any GNU system.

A GUI for ps(1) Built with Mozilla *by Nigel McFarlane*

Make your apps run anywhere your browser does with the development framework that's already on your desktop.

Indepth

Eclipse Goes Native *by John Healy, Andrew Haley and Tom Tromey*

Now you don't have to wait for a JVM to run your Java app on a new platform.

Clusters for Nothing and Nodes for Free *by Alexander Perry, Hoke Trammell and David Haynes*

The processing power you need for big nightly jobs is all around you, and the desktop users won't miss it.

LINUX JOURNAL

Rapid Application Development with Python and Glade

David Reed

Issue #123, July 2004

Create and modify your Python application's GUI using the easy design tool Glade; then, automate the process of setting up event handlers.

Writing GUI programs involves two basic steps. First, you need to write the code to create the interface, with elements, such as menus, and widgets, such as buttons, labels and entry fields. You then need to write the code that executes when events occur, such as when a button is pressed or a menu item is selected. When the program runs, it enters an event loop that repeatedly waits for an event and then calls the event handler, also known as a callback function, that was defined for that event. For example, you write a function to be called when a button is pressed. Writing code to display the widgets, defining the functions to be called when events occur and connecting each event to the specific function is a tedious process to do by hand.

For many widget sets, programs exist to lay out the GUI visually. Damon Chaplin wrote the Glade program to allow users to create an interface visually using the GTK/GNOME widgets and also to specify which functions to call when events occur. Glade stores the layout of the widgets and the callbacks as an XML file. Glade also generates a C or C++ program that contains all the calls to create the widgets in the specified layout, connect the callbacks and define empty functions for each callback. However, Glade does not create Python code. The GladeGen program I wrote generates Python code based on the Glade XML file.

If you change your GUI using Glade, you need Glade to output the new C/C++ code to create the GUI. This can be annoying, especially if you have modified the code that creates the GUI. James Henstridge wrote libglade to alleviate the need to hard code the GUI-generating code in your program. James also started and helps maintain the GTK/GNOME Python bindings, a Python module that provides access to GTK/GNOME C functions. With libglade, your program does not contain code to create the interface. Instead, libglade parses the XML file

when your program is run and creates the interface on the fly at runtime. Thus, whenever you change your GUI using Glade, you do not need to change the code that creates the interface.

I prefer using Python except in cases of code containing a lot of computation where speed is crucial. Python's high-level data structures and interpretative environment make it quick to develop, modify and maintain code. The September 2003 issue of *Linux Journal* contains an introductory article on PyGTK and Glade (see the on-line Resources section) that readers unfamiliar with Gtk and Glade will find helpful.

The motivation for GladeGen was a patient database/accounting system I was writing for the optometric/optical offices where my wife works. Before my wife started working with them, they were using a Microsoft Access database system someone had written. That person had moved out of state, and the office wanted a new system. They talked to other optical offices around town to find out what software was being used. People at each office complained about the software; the systems were buggy and expensive. I convinced the owner of my wife's office that I could write a custom system during the summer for about the same cost as other new software, and that it would do exactly what they needed it to do. But, he had to let me use Linux.

The end result is a Python/GTK program that uses PostgreSQL as the back-end database to store all of the data. This allows all of the searching and tabulating to be done by SQL commands. The Python code provides the layer of code that communicates between the PostgreSQL database and GTK interface. Both GTK and PostgreSQL are written in C, so they run fast. The Python code is more than fast enough on modern processors for handling the communication between GTK and PostgreSQL. The PyGTK front end and PostgreSQL database allow the client front end to access any database server so they can run the front-end GUI on multiple computers and access the database server. The client/server setup allows them to run the front-end GUI and access the PostgreSQL server at other locations over the Internet through an SSH-encrypted connection. It also allows me to have remote access from home when users have questions about the system.

The program has more than 40 windows, including those for entering patient information, frame/lens purchases, contacts, reconciling insurance payments and entering and tracking the frame inventory. I decided to use Glade to create each window, and because I wanted to use Python, I needed to write my own software to automate code creation for each window. The result is GladeGen.

GladeGen Usage

The code I have written automates the task of creating Python code to create the interface, connect the callback functions, provide access to the widgets and create empty callback functions based on the Glade XML file. Using this software, the steps for creating a GUI program are 1) use Glade to make the interface visually and save the XML file, 2) run GladeGen to generate the code and 3) write the code for the callbacks.

The code GladeGen creates contains all the code to run the application and display your interface, along with empty callback functions. It also allows you to use Glade to modify the interface. When you rerun GladeGen, it regenerates the application code with any additional callbacks and new widgets, without changing or modifying any of the existing code.

Here, I demonstrate how to use GladeGen by creating a math quiz program. The complete program is available from the *Linux Journal* FTP site (see Resources). GladeGen works with the GTK 2.x widget set and the corresponding version of Glade, which on Red Hat systems is named glade-2. If you are familiar with the GTK widgets, Glade is fairly intuitive to use. If not, you should familiarize yourself with the various container widgets, including table and horizontal/vertical box.

Using glade-2, I created the first version of the interface. I started with a GtkWindow and added a GtkVBox. I placed two GtkFrame widgets in the vertical box and a GtkTable in each frame. All the other widgets are placed in the two table widgets. I used GtkSpinButton widgets to allow the user to select the number of digits and operators in the problem. GtkCheckButton widgets are used to indicate which operators should be included in the problem. GtkEntry widgets are used for the problem, the answer and information on the number of correct/incorrect problems answered. The other widgets are GtkLabel and GtkButton widgets.

I saved the Glade file as mathflash.glade. Glade does not ask if you want to save your file, so you need to remember to save it before quitting if you make any changes to your interface. See Figure 1 for a screenshot of the interface and Glade. It shows the reset_button is configured to call the function on_reset_button_clicked when the button is clicked.

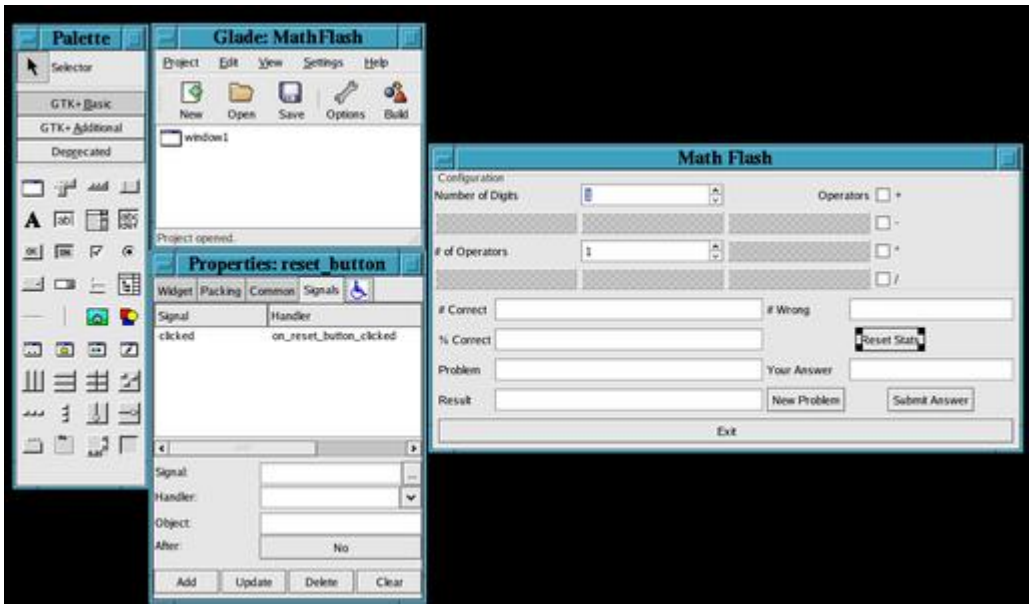


Figure 1. Setting the Callback for a Button in Glade

Glade allows you to give each widget a name or provides a default name. For the widgets we need to interact with, such as buttons and data entry widgets, I provided names that match their intended uses. With Glade, you also specify the signals you want to connect to callback functions. As mentioned above, using Glade, I specified that on_reset_button should be called when the reset_button is clicked.

Next, I used GladeGen to produce template code for the application using the command `GladeGen.py mathflash.glade MathFlash MathFlash`. The command-line arguments are the name of the Glade XML file, the name of the Python file/module to create and the name of the class to create in that file/module. The resulting MathFlash.py file can be found in Listing 1. The MathFlash class subclasses a GladeWindow class, the class that uses libglade to connect all the callbacks listed in the handlers variable. It also creates a dictionary, self.widgets, that maps each widget name in the widget_list variable to the corresponding GtkWidget instance. The GladeWindow subclass provides default show and hide methods so that the template code can be run immediately to view the interface before you start writing the callbacks.

Listing 1. GladeGen produced this Python code from the XML file created in Glade.

```
#!/usr/bin/env python

#-----------------------------------------
# MathFlash.py
# Dave Reed
# 02/28/2004
#-----------------------------------------

import sys

from GladeWindow import *
```

```
#-----------------------------------------

class MathFlash(GladeWindow):

    #-----------------------------------

    def __init__(self):

        ''' '''

        self.init()

    #-----------------------------------

    def init(self):

        filename = 'mathflash.glade'

        widget_list = [
            'window1',
            'plus_check',
            'minus_check',
            'multiply_check',
            'divide_check',
            'digits_spin',
            'operators_spin',
            'correct_entry',
            'wrong_entry',
            'pct_entry',
            'problem_entry',
            'answer_entry',
            'submit_button',
            'reset_button',
            'exit_button',
            'new_button',
            'result_entry',
            ]

        handlers = [
            'on_operator_check_toggled',
            'on_submit_button_clicked',
            'on_reset_button_clicked',
            'on_exit_button_clicked',
            'on_new_button_clicked',
            ]

        top_window = 'window1'
        GladeWindow.__init__(self, filename,
                             top_window, widget_list,
                             handlers)

    #-----------------------------------

    def on_operator_check_toggled(self, *args):
        pass

    def on_submit_button_clicked(self, *args):
        pass

    def on_reset_button_clicked(self, *args):
        pass

    def on_exit_button_clicked(self, *args):
        pass

    def on_new_button_clicked(self, *args):
        pass

#-----------------------------------------

def main(argv):

    w = MathFlash()
    w.show()
    gtk.main()

#-----------------------------------------
```

```
if __name__ == '__main__':
    main(sys.argv)
```

The GladeGenConfig.py file allows customizations to specify the program author, the widget types you want put in the self.widgets dictionary and how the created Python code file should look. In the GladeGenConfig.py file provided, the widget types GtkWindow, GtkButton, GtkSpinButton, GtkCheckButton, GtkEntry, GtkCombo and GtkTextView are listed. It rarely is necessary to access a GtkLabel widget and the container widgets, so I have not included them in the include_widget_types list in GladeGenConfig.

The created MathFlash.py file contains methods for each of the callback functions with the Python no-op statement pass. The method is declared with the self parameter and *args for a variable length parameter list. In Python, *args allow the function/method to be passed as many parameters as the caller wants, and they are received in the calling function as a tuple. Most of the callbacks are passed the widget that the event occurred in and sometimes additional parameters specific to the event. The API reference available on the GTK Web site shows the exact parameters for each callback (see Resources). Now, we need to add code to the callbacks and any other code the program needs. For this program, about 60 additional lines of Python code result in the final MathFlash.py file. An experienced Glade user and Python programmer should be able to create the entire program from scratch in less than 30 minutes. For more complicated programs, the Model/View/Controller design pattern could be used. The code GladeGen produces would play the role of the controller.

Let's examine the callback on_submit_button_clicked to understand the details of how to use the PyGTK code GladeGen produces. Here is the Python code I wrote:

```
def on_submit_button_clicked(self, *args):

    prob = self.widgets['problem_entry'].get_text()
    ans = eval(prob)
    user = int(self.widgets['answer_entry'].get_text())
    self.total += 1
    if ans == user:
        self.correct += 1
        self.widgets['result_entry'].set_text('Correct')
    else:
        self.widgets['result_entry'].set_text(
            'Wrong, the answer is %d' % ans)
    self.show_results()
```

The C GTK API contains the function G_CONST_RETURN gchar* gtk_entry_get_text(GtkEntry *entry). Because Python is an object-oriented language, the bindings are set up as methods for the class. For the Python bindings, the gtk_ and widget name prefixes are removed from the name of the function and called with a Python instance of that widget. This same pattern

applies to all of the GTK functions. Using the self.widgets instance, the corresponding Python call becomes `self.widgets['problem_entry'].get_text()`, where problem_entry is the name for the entry widget in the Glade XML file.

I then used the Python eval function to determine the answer to the problem. This is much simpler than writing my own parser to evaluate the expression. The usual rule that multiplication and division have a higher precedence than addition and subtraction applies. Use of the eval function can be a security issue if you allow the user to enter the string that is evaluated, but in this case our program is producing the string that is evaluated, so we do not need to worry about it.

If you modify the Glade XML file, you can rerun GladeGen and it will add any new callback methods without overwriting or losing any code you have written other than the init method, which you should never modify. GladeGen produces a new init method each time you run it. The init method contains the names of the widgets and callbacks, so anytime the Glade file is updated, it needs to be reproduced. In my case, I later added a button to reset the stats. When I reran GladeGen, it added an empty on_reset_button_clicked method and provided a new init method that listed the reset_button widget and on_reset_button_clicked callback. Because I am using libglade to create the interface at runtime, no additional modifications are necessary.

How GladeGen Works

GladeGen first determines whether the specified module/file exists; if not, it creates a basic file with author information and a class definition. GladeGen then parses the Glade XML file and finds a list of widgets and handlers. Python provides the inspect module that allows a program to determine what functions, classes and methods an existing Python module contains and which lines correspond to each. GladeGen uses the inspect module to determine which callbacks have been written already so that they are not replaced with an empty callback method. GladeGen adds any new callbacks to the bottom of the class definition. The inspect module also allows GladeGen to determine which lines contain the init method and to replace them with a new init method containing all the widgets and handlers in the latest Glade XML file.

Python supports both the standard DOM and SAX interfaces for parsing XML files. The SAX interface is an event-driven model in which the user sets up functions to be called as XML tags are processed. The DOM interface reads the entire XML file into memory and provides functions for traversing the XML hierarchy and retrieving the information. For GladeGen, we wanted to extract only certain information from the XML file, so the DOM interface is simpler to use. Also, the size of a Glade XML file is small enough that reading the entire file

into memory and generating the Python representation of it should not require a large amount of memory. Using the DOM interface, the get_xml method in the GladeGen class extracts the widget names and handler names from a Glade XML file using about 30 lines of Python code.

Summary

Glade and GladeGen automate much of the tedious, repetitive work that goes into creating graphical programs by removing the need to write code to create and store widgets and set up the callback functions. This allows for rapid application development of Python/GNOME/GTK applications. The finished Math Flash is shown in Figure 2. The GladeGen software can run on any system that supports Python and GTK, including Linux, UNIX, Mac OS X and Microsoft Windows.

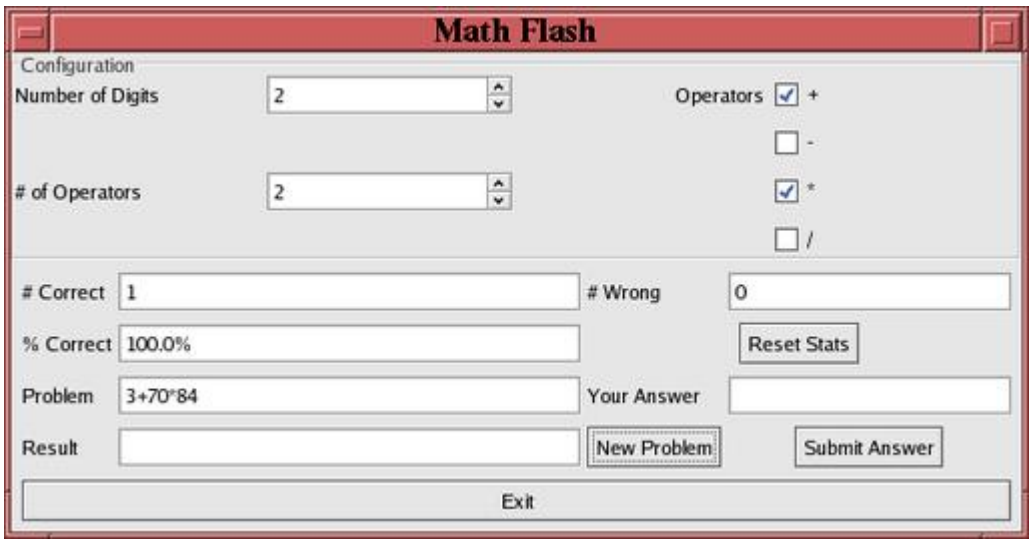


Figure 2. Math Flash

A number of features could be added to this system. Instead of using the generic *args parameter for the created callback functions, the parameters could be specified explicitly, based on the widget and callback prototype. I also plan to add a graphical front end to the program for configuring the options in the GladeGenConfig.py file. The GladeGen software is released under the GPL. If anyone is interested in modifying/extending it, please let the author know.

Acknowledgements

Thanks to one of my students, Jeremiah Schilens, who worked on an earlier version of this project with me.

Resources for this article: /article/7558.

David Reed lives in Columbus, Ohio with his wife and two dogs. He has worked with UNIX systems since 1991 and Linux since 1997. He holds a PhD in

volumetric graphics from The Ohio State University and currently teaches computer science at Capital University. Capital uses a mixture of Python, C++ and Java throughout its CS curriculum. David can be reached at dreed@capital.edu.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Cross-Platform Network Applications with Mono

Ian Pointer

Issue #123, July 2004

Curious about .NET? Try out this useful sample app that exercises the GUI and XML-RPC features of Mono.

Mono is Ximian's open-source implementation of Microsoft's .NET development framework. .NET contains several different technologies: a set of compilers for many different languages (including Microsoft's new language, C#) that generates platform-independent bytecode; a virtual machine known as the Common Language Runtime (CLR) that runs these bytecodes; and a class library full of useful programs for performing actions ranging from file I/O to GUI creation and operation.

The Mono implementation includes a CLR that runs on Linux, BSD-based systems (including Mac OS X) and Windows, plus compilers for C# and Basic. Mono is a work-in-progress, and many parts of the .NET class library have yet to be implemented, specifically the Windows.Forms group that contains classes for working with the Windows GUI. However, the Mono developers have released bindings for the GTK user-interface toolkit, so cross-platform graphical applications can be constructed even without 100% .NET compatibility. This article describes how to use C#, Mono and Linux to write a useful program, MonoBlog, that can run on any system that runs Mono and GTK. Some familiarity with Glade and C# is assumed, but only at a basic level. Helpful tutorials can be found in the on-line Resources section.

Obtaining Mono

The Mono Web site has instructions for installing the system on Linux, Mac OS X and Windows platforms (see Resources). You also need two additional C# libraries, GTK# and XmlRpcCS. The systems that MonoBlog runs on require the base GTK libraries, which are available on most Linux systems. They probably need to be installed on Windows and Mac OS X systems, however; packages

can be found on the GTK Web site (see Resources). Instructions for installing these libraries can be found on their respective Web pages.

MonoBlog, a Weblog Editor

MonoBlog is a Weblog editor that can add new posts to a Weblog and edit old ones, as well as provide a way for a user to change configuration settings. Most Weblog systems implement a common base of functionality known as the MetaWeblog API. MonoBlog uses this to communicate with a variety of different Weblog programs, rather than write a separate back end for Movable Type, LiveJournal or Radio Userland systems. The complete C# code for this example is available on the *Linux Journal* FTP site (see Resources).

Figures 1 and 2 show the user interface of MonoBlog, created using Glade on Linux. The main window in Figure 1 has text controls for entering Weblog titles and the content, plus a series of buttons for updating the Weblog, clearing the forms and quitting the program. The expanse of white on the left-hand side is a GTKTreeView control, which displays a list of older posts the user can click on in order to update. The window shown in Figure 2 is a simple preferences panel where users enter the information that allows MonoBlog to communicate with their Weblogs.

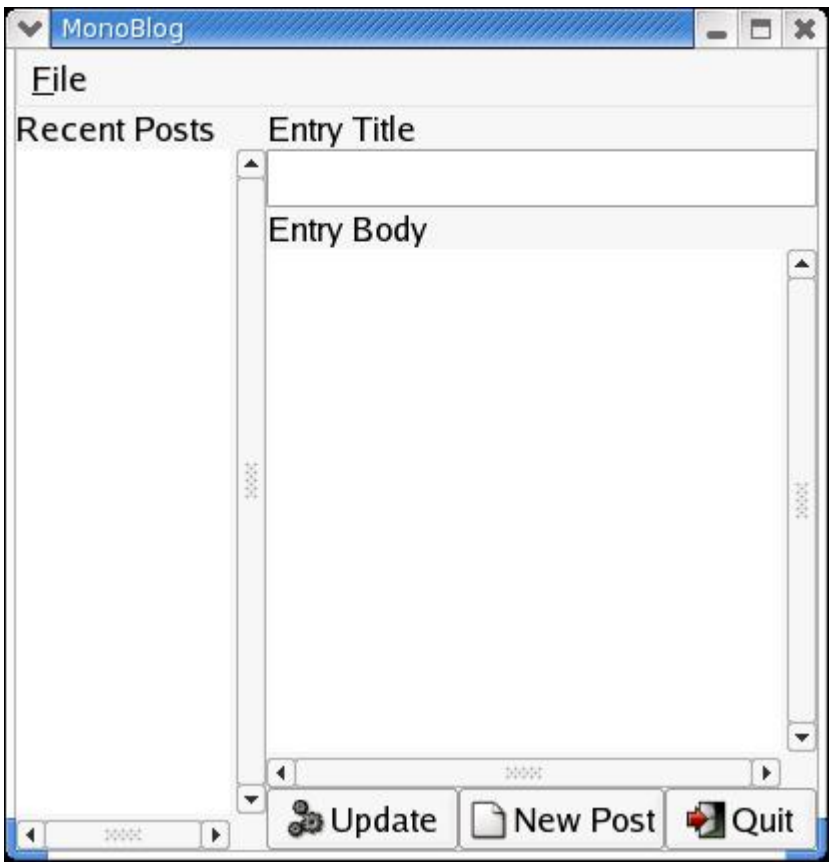


Figure 1. The Main Window

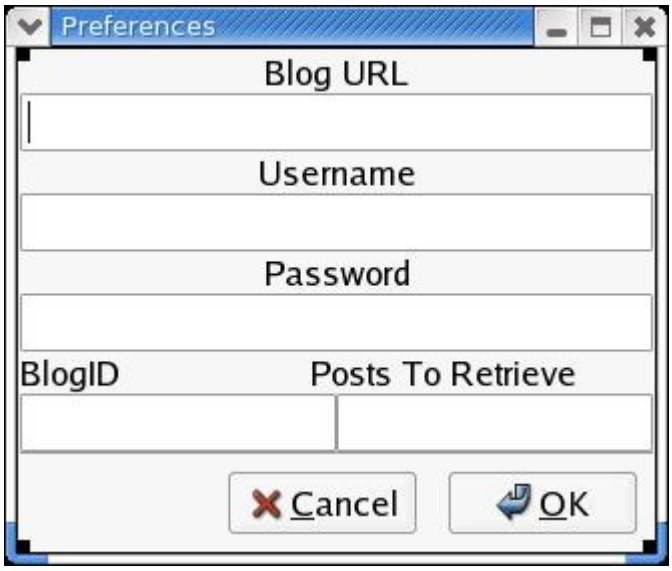


Figure 2. The Preferences Window

Creating the GUI with libglade

One of GTK's useful features is libglade, a library that allows us to construct a program's GUI by reading in the XML files created by Glade, specifying the layout of the widgets in the code itself. The GTK# binding includes this functionality, so building the GUI is quite easy. At the start of MonoBlog, we import the GTK and Glade namespaces with the `using` statement. Then, in the constructor, we have:

```
Application.Init();

Glade.XML gxml = new Glade.XML("monoblog.glade",
                               null, null);
gxml.Autoconnect(this);

Application.Run();
```

The calls to the Application class are required in all GTK programs. Application.Init() performs GTK initialization, and Application.Run() passes control of the program to the GTK main loop, which watches for events and reports signals back when they occur. The standard Glade.XML constructor takes three arguments: a string containing the filename of the Glade file, a string that tells the object the node in the Glade tree where it should start building the interface and, finally, a string that can be used to specify a translation domain for the Glade file in question.

MonoBlog needs to have access to all the nodes in the XML file, both the main window and the preferences panel. No translation is required, so the second and third arguments are null. The Autoconnect() method binds the object given as an argument with the signal handlers and objects defined in the Glade file, allowing that object to respond to events and manipulate widgets. As MonoBlog is a small program, I have contained all the signal handling within

the main class. In larger, more complex systems, it might be advisable to separate signal handling out into another class.

To access the widgets, a special declaration is required. The widget must be declared as an instance variable, using a custom attribute:

```
[Glade.Widget] GTKWidgetType widgetname;
```

with `GTKWidgetType` being replaced by the actual object type concerned, and `widgetname` being the name of the widget as defined in the Glade file. After Autoconnect() returns, these widgets can be used as if they had been created in the program itself.

Retrieving Old Entries

When the program loads, the first action it performs is to query the Weblog and download the recent posts, displaying them in the TreeView widget. The method getRecentPosts() in the MonoBlog class handles this; it is called in the main constructor if preferences exist, as the method needs to know about the Weblog it is contacting. The MetaWeblog API provides a function call, metaweblog.getRecentPosts, that returns a specified number of old posts or as many as it can find if fewer posts exist than we desire.

The communication with the Weblog is straightforward:

```
XmlRpcRequest client = new XmlRpcRequest();
client.MethodName = "metaWeblog.getRecentPosts";
client.Params.Add(BlogID);
client.Params.Add(ServerUser);
client.Params.Add(ServerPass);
client.Params.Add(NumberOfPosts);
XmlRpcResponse response = client.Send(ServerURL);
```

All that is required is to create a new XmlRpcRequest object, set the required API method name, fill in the necessary arguments and send it to the Weblog. The Weblog then returns a response, in this case an array of posts, which is stored in the Value field of the XmlRpcResponse object. Next, we need to update the GTKTreeView control.

In GTK 2.0 and above, this control uses a model-view-controller approach. Here, then, we create a new model object and pass it to the control:

```
System.Type[] ListTypes = new System.Type[3];
ListTypes[0] = typeof(string);
ListTypes[1] = typeof(string);
ListTypes[2] = typeof(string);
ListStore store = new ListStore(ListTypes);
treeview1.Model = store;
```

This model object creates a table with three columns. The ListStore object needs to be passed an array of Type objects; each item in the array corresponds to the type of column. A Weblog post contains three items—a title, the content of the post and a unique identifier—all of which are strings, so here all the columns have a String type. The rest of the method cycles through the array, populating the model:

```
TreeIter iter = new TreeIter ();
foreach (Hashtable post in results) {
  String title = (String) post ["title"];
  String postid = (String) post ["postid"];
  String description = (String) post ["description"];

  store.Append (out iter);
  store.SetValue (iter, 0, new GLib.Value(title));
  store.SetValue (iter, 1, new GLib.Value(postid));
  store.SetValue (iter, 2,
  new GLib.Value(description));
}
```

This, by itself, isn't enough to display the titles in the tree. For that, we include some code in the constructor, after the call to getRecentPosts():

```
TreeViewColumn TitleCol = new TreeViewColumn();
CellRenderer TitleRenderer = new CellRendererText();
TitleCol.AddAttribute (TitleRenderer, "text", 0);
treeview1.AppendColumn (TitleCol);
```

This adds a new column view to the tree. The AddAtrribute() method is hooked to the title column of the model (the first) with the 0 argument. As all the user requires is to see the title of an entry in the TreeView control; no other column views are needed. The information, though, is stored in the model to make the program more efficient.

Editing Old Posts

When a user clicks on an entry, the desired result is for the program to display the old entry in the text entry portions on the right-hand side of the window. The MetaWeblog API has a method called metaweblog.getPost that pulls posts from the Weblog. As they already have been downloaded in the getRecentPosts() method, the program can get the data from the model instead of communicating with the Weblog again. The row_activated signal is bound to the method SelectOldPost using Glade, so whenever an item is double-clicked, this code runs:

```
public void SelectOldPost(System.Object obj, EventArgs e) {
  TreeSelection currentSelection = treeview1.Selection;
  TreeIter iter;
  TreeModel model = treeview1.Model;
  currentSelection.GetSelected (out model, out iter);
  String selected = (string) model.GetValue (iter, 1);
  String oldTitle = (string) model.GetValue(iter,0);
  String oldEntry = (string) model.GetValue(iter,2);

  TextBuffer buffer = textview1.Buffer;
  entry1.Text = oldTitle;
```

```
    buffer.SetText(oldEntry);

    OldPostID = selected;
    EditingOldPost = true;
  }
```

This method obtains the current selected item in the GTKTreeView control and uses an iterator to index into the model and find the required values. It then fills in the text fields with this information and updates two instance variables that are needed when the user clicks on the Post button. If the program is updating an older entry rather than creating a new entry, it needs to make a different MetaWeblog API call, which needs the unique identifier of the post. The variables OldPostID and EditingOldPost are updated to reflect this.

Updating the Weblog

The clicked signal on the Update button is bound to the method OnUpdateClicked. This process is too long to reprint in full in this article, but the operation is simple enough. First, it gets the text from the two text controls and creates a hash table representation of the post; this is required for the MetaWeblog API call. Depending on whether the EditingOldPost flag is set, the method then sends an XML-RPC request to the Weblog, using the `metaweblog.newPost` or `metaweblog.editPost` calls as appropriate. When the Weblog returns a successful response, indicating that it has been updated, the method finally clears the text forms and allows the user to start anew.

The other buttons on the main window, New Post and Quit, are short snippets of code. Like the Post button, the clicked signals are bound in MonoBlog. New Post is bound to a method that clears the text forms and sets the EditingOldPost flag to false, allowing the user to start over. Quit, as expected, exits MonoBlog using the Application.Exit() GTK call.

Preferences

The .NET class library includes classes that handle reading in preferences from an XML file. Listing 1 shows an example MonoBlog configuration file.

Listing 1. An Example XML Configuration File

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<appSettings>
<add key="ServerURL"
value="http://www.test.com/mt-xmlrpc.cgi"/>
<add key="ServerUser" value="example"/>
<add key="ServerPass" value="password"/>
<add key="BlogID" value="1"/>
<add key="NumberOfPosts" value="10"/>
</appSettings>
```

```
  </configuration>
```

The method getConfig(), shown below, reads in these values:

```
private bool getConfig {
  try {
  AppSettingsReader config = new AppSettingsReader();
  ServerURL = (string) config.GetValue("ServerURL",
  typeof(string));

  ServerUser = (string)  config.GetValue("ServerUser",
  typeof(string));

  ServerPass = (string) config.GetValue("ServerPass",
  typeof(string));

  BlogID = (string) config.GetValue("BlogID",
  typeof(string));

  NumberOfPosts =  (string)
  config.GetValue("NumberOfPosts", typeof(string));

  catch(Exception problem) {
    return false;
    }
  return true;
}
```

The AppSettingsReader object, by default, looks for a configuration file named executable.config, so here it opens a file called monoblog.exe.config. Then, the GetValue() method is used to determine the required preference values. MonoBlog calls this method in its constructor before it attempts to query the Weblog for old posts, so it has the required information. If the file does not exist or if there is a problem reading the data, the method returns false. The constructor only calls getRecentPosts() if this method returns true, preventing garbage values from being used.

Updating the preferences is a more complex task. First, the Preferences option in the main window's menubar is bound to the method OnPrefsActivate, using Glade's Menu Editor. This brings up the dialog shown in Figure 2 and fills in the fields with the current values, if any are defined. When the user clicks on the OK button in this dialog, MonoBlog updates the variables and writes the new information back out to the configuration file. Unfortunately, the .NET class library doesn't have classes that update configuration files. As the configuration here is fairly simple, I wrote a method called saveConfig() that opens the default configuration file and writes the updated information back out to disk using a series of Write() statements. This could be replaced with a more sophisticated method that builds a proper XML document, but it was easier for this application simply to write out the values in a plain manner.

Error Handling

As MonoBlog makes calls to the Internet where things can go wrong that aren't within the control of the program (network errors, name server problems and

so on), it contains some basic error handling functionality. The getRecentPosts() and OnUpdateClicked() methods are wrapped in a try...catch block. The code that accesses the Internet is executed, and if there is a problem, the following catch block runs:

```
catch(Exception problem) {
  MessageDialog md =
  new MessageDialog(MonoBlogWindow,
  DialogFlags.DestroyWithParent,
  MessageType.Error,
  ButtonsType.Close,
  problem.ToString());

  md.Run();
  md.Destroy();
}
```

This causes an Error dialog to appear on screen, with the problem as reported by the Mono CLR included as a text message. This allows the user to continue and possibly fix the problem. However, at the time of this writing, exception support is not working in the PPC branch of the Mono CLR, so if the program runs on Mac OS X, the exception mechanism does not work and the program fails silently. Work is proceeding on the PPC port, though, so by the time this article makes it to print, this lack of support may no longer be an issue.

Compiling and Running

Compiling C# programs is done by way of the mcs compiler. MonoBlog is compiled with this command:

```
mcs -r gtk-sharp.dll -r glade-sharp.dll \
-r XmlRpcCS.dll -r glib-sharp.dll monoblog.cs
```

The -r option indicates a resource the program needs; here, we simply need to specify which libraries MonoBlog uses. This produces a compiled bytecode file called monoblog.exe. To run the program, we need to run the Mono CLR with this file as a parameter:

```
mono monoblog.exe
```

Now, having developed the program on Linux, we can run the program on Windows or Mac OS X with a minimum of fuss. Simply copy monoblog.exe, monoblog.exe.config and monoblog.glade files to the other platform and run them using the Mono CLR, as shown above.

Figures 3, 4 and 5 show MonoBlog running on Linux, Windows and Mac OS X machines, respectively. No code has to be changed; the program works as is, as long as all the libraries MonoBlog uses are present.

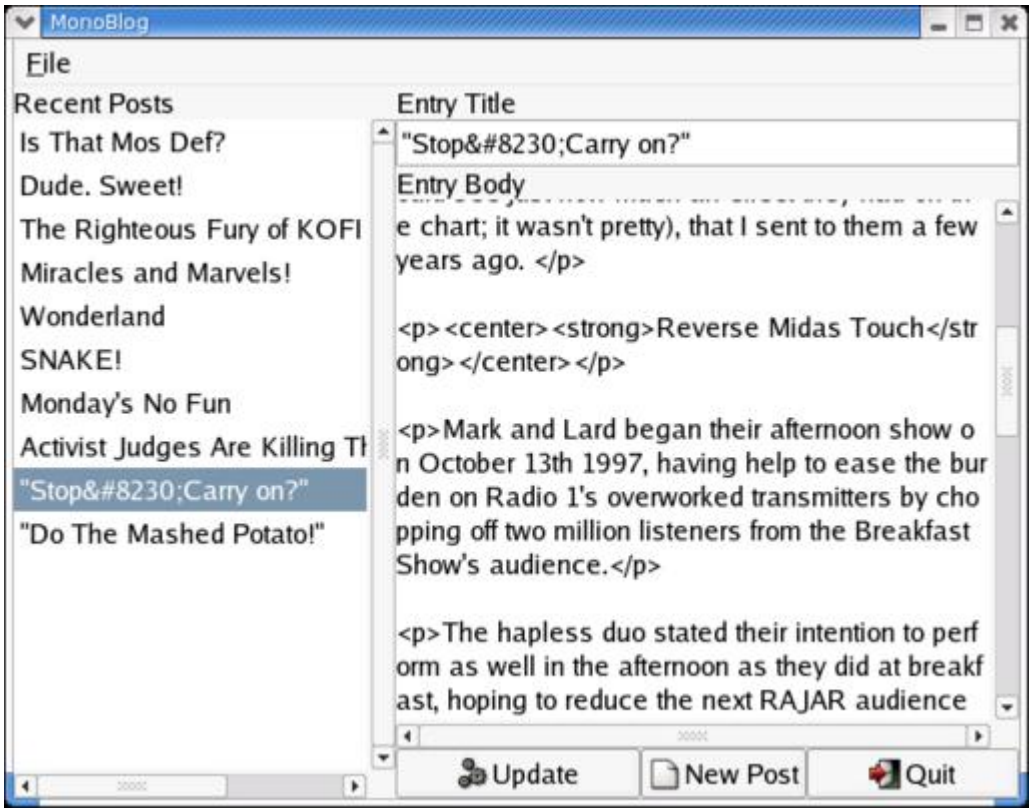


Figure 3. MonoBlog on Red Hat 9

Figure 4. MonoBlog on Windows XP

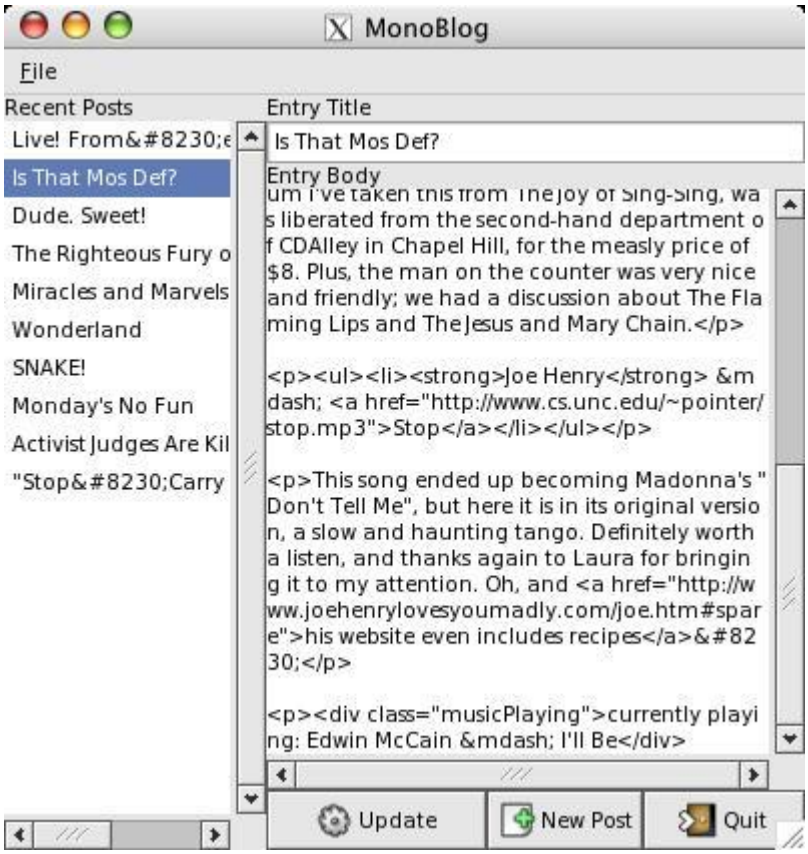


Figure 5. MonoBlog on Mac OS X

Conclusion

Hopefully, this article has demonstrated how Mono and C# can be used to create cross-platform applications quickly and easily. You can develop on one platform and be assured that the program can run on any platform that runs Mono and the GTK libraries. The MonoBlog program itself is ripe for further experimentation. Some possible areas for improvement are extra formatting options, more detailed error reporting, using the GtkHTML# bindings to create an HTML preview window, and further implementation of the MetaWeblog API, such as adding the ability to delete posts from a Weblog.

Resources for this article: /article/7557.

Ian Pointer is an unemployed Computer Science graduate in the UK. He can be reached at ian@snappishproductions.com.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Developing for Windows on Linux

Joey Bernard

Issue #123, July 2004

Build and test both the Linux and Microsoft Windows versions of your project without rebooting. With the free tools MinGW and Wine, you might even call Win32 a cross-platform API.

Like most people who read *Linux Journal*, I am a rabid fan of all things Linux, GNU and open source. I run Linux on all of my personal machines, program on them, play on them and evangelize to others whenever possible. But, a large portion of the programming jobs out there involve writing applications for an operating system from Redmond, Washington.

For my job, I've had to write some smaller applications for the Microsoft Windows platform. Because speed of execution was an issue, I was going to have to write them in C, directly using the Win32 API. It occurred to me that if I was going to be using a standard language such as C, I might be able to develop in my nice and cozy Linux home.

This article is a short guide on developing an application for Windows completely within a Linux environment. I give a short introduction to Windows programming and step through compiling and testing a sample program. I also discuss Wine to simplify porting Windows source code to Linux.

Win32 Programming

For those of us raised on the wholesome nutrition of a UNIX-style process abstraction, the Windows model might seem downright heretical. The Windows model is a preemptive, multitasking, multithreaded, message-passing operating system. I'm limiting myself here to NT and its derivatives, 2000 and XP. All processes are considered threads by the operating system. This makes the process context slightly lighter than the traditional heavyweight process model used in UNIX-like operating systems. As a consequence of this everything-is-a-thread model, however, everything sits in global memory address space. With

the correct permissions and the correct address, one program could twiddle another program's bits.

Another consequence of this is data structures created by the kernel don't sit at any fixed address. This means it is up to the user program to lock down the associated memory before using any global data structures, such as graphic contexts. You also must remember to unlock these structures once you are done with them, or they may help cause memory fragmentation.

Listing 1, available from the *Linux Journal* FTP site (see the on-line Resources section), is a basic Hello World program. Most of it is boilerplate, and only the portion within the switch statement is of any real interest. It does seem like quite a bit of code for a basic program, but that is the problem with using a low-level API. A good comparison on Linux would be writing code for X using Xt.

Instead of a main() function, a Windows GUI program starts at WinMain(). It's in this function that your program does all of its initialization. Part of this initialization includes defining the window class for the main window and associating a callback function for it. Next, create the main window and show it on the desktop. Control then passes to the message loop, and the callback function processes the messages that are sent to the main window.

A good quick introduction to writing Windows programs is available from winprog.org (see Resources). The authors of this Web site offer a good FAQ and a fairly good tutorial covering all of the basics. Of course, the bible for Windows programming is the massive book *Windows Programming*, by Charles Petzold. If you can't find what you need in this tome, you always can use it to beat the information out of your friendly neighbourhood Windows guru.

Cross-Compiling

One of the amazing things about GCC is that it has been ported to so many different platforms and operating systems. A great gift that comes from this is the ability to compile binaries on one platform that are destined for a completely different one. I regularly compile binaries for Solaris or Windows on my Linux laptop. This is an amazing advantage, allowing development to occur in a familiar, comfortable environment.

The purest way to set up is to go back to the source (see Resources). This way you can compile code with the exact settings and for the exact platform you want. Thankfully, this work has already been done. The good people at the MinGW Project maintain a port of GCC for compiling Windows binaries. This includes all of the associated files, such as the headers. The sources are available here along with binary tarballs. These programs also have been packaged for RPM-based and deb-based distributions. If you are running

Debian, you can use apt-get to retrieve the mingw32 and mingw32-runtime packages. If you are running testing or unstable, you also should grab mingw32-binutils.

Most of the compilation options in GCC are available here in MinGW, along with a few extras. If you simply compile a program without any extra options, it can be run from the console. This is what you would do if you wanted to write a small, simple program that did not need a GUI. Because this is Windows, we want a GUI program, so we write all of the required boilerplate we saw above and add the `-mwindows` option to the compilation command. This sets up the macros and library links you need in order to compile a standard Windows executable. If you decide to write a more complicated Windows program that uses some other Windows' feature, you need to add in the libraries explicitly that you want linked.

In Windows you can define resources for your program. These include such items as menus, bitmaps and text strings, among others. These resources are defined in a separate file and need to be compiled separately before being linked to your executable. That job falls to the program mingw-windres, which creates an object file you subsequently can link to your executable.

To compile our simple example program shown in Listing 1, we use the command:

```
mingw-gcc -o example1.exe example.c -mwindows
```

Replace the command `mingw-gcc` with whatever the package maintainer called the compiler executable for your package. Presto, you now have a Windows program ready for the world. Or is it?

Debugging with Wine

Wine is the other great boon for developers who need to write programs for the Windows platform. The massive amount of work that has gone into Wine by its developers is phenomenal. This great program allows you to run Windows programs from within Linux. The upshot to this is we now can run our freshly compiled program and see if it actually works as advertised. To do this, use the command `wine example1.exe`, and you should see the window appear on your desktop (Figure 1). When you set up Wine, you have the options of windows being managed by your window manager, not managed or displayed on their own desktop. This affects how the window looks when you run it. What you see in Figure 1 is an unmanaged application.

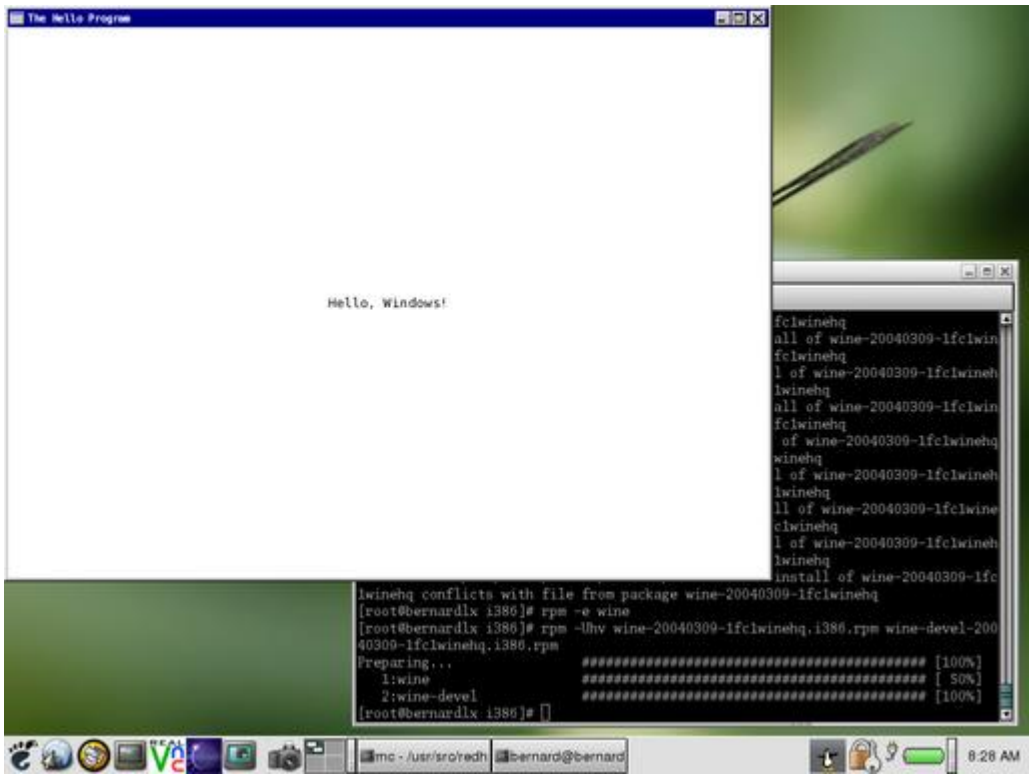


Figure 1. The Example Application Running Unmanaged under Wine

If you weren't lucky enough to have typed your program perfectly, you may need to do some debugging to figure out what has gone wrong. Wine can be a great asset here. The option --debugmsg [debugchannel] outputs the results from one or more debug channels within Wine. Examples of the available debug channels are:

- relay: writes a log message every time a Win32 function is called.
- win: tracks Windows messages.
- all: tracks all messages.

Don't use all unless you really need it. The amount of output quickly can overwhelm even the most detail-obsessed programmer. A complete list of available debug channels can be found on the Wine site.

Compiling a Native Version for Linux

We now have a wonderful, working, bug-free program that runs under Windows. Considering that all of the work was done under Linux, wouldn't it be nice if we also could have our program run under Linux? The good folks at the Wine Project have come to the rescue again. Part of the project includes winelib, a library that provides the interface to Linux for your Windows source code. In order to use this functionality, you need to install the wine-devel package for your distribution. If you installed from source, the required files already should be available.

Included in the wine-devel package is a Perl script called winemaker. This script is designed to go through your source files and directories and make the required changes to get it to compile correctly against winelib on Linux. Things it checks include filename case and line ending characters. In addition, it replaces file path back slashes with forward slashes and does many other things. By default, it backs up any source files it needs to change. It converts your project to winelib, making all kinds of automagic changes. To compile, you simply run:

```
winemaker .
./configure --with-wine=/path/to/wine
make
```

to create a Linux executable. The dot you see above is there on purpose. You hand in the path where winemaker can find the source files it needs to analyze; here, the files are in the current directory.

In our case, our sample doesn't have any project files, and winemaker thinks this is a bit of a problem. We can do the steps involved simply by hand. Instead of executing `mingw-gcc` to compile our program, we use `winegcc` with the exact same command-line parameters. This creates a file ending in .so and a shell script to handle the program execution. We now have our Windows source code compiled and running under Linux.

Conclusion

I hope I've been able to convince some of the Windows developers out there that they can work effectively from within Linux. With GCC allowing compilation of an executable for Windows, and Wine providing great support in running and debugging, there is no real reason to boot up Windows in most cases. The only reason would be if your favourite IDE didn't run correctly under Wine, but then you always could volunteer to fix that problem, right?

As this was only a short introduction, I did not cover support for MFC or the creation of DLLs. Both of these topics are discussed in more detail at WineHQ and the MinGW site.

Resources for this article: /article/7555.

Joey Bernard is a systems architect for CARIS, a GIS company in Canada. He's never actually done any GIS work, mostly just Oracle, UNIX systems programming and some Windows programming.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

A GUI for ps(1) Built with Mozilla

Nigel McFarlane

Issue #123, July 2004

Give your command-line tasks a GUI with the Mozilla platform.

One of the more powerful features of Linux is the simple way that new commands can be constructed using aliases, shell scripts and other textual tricks. These techniques rely on a command-line interface, but what if you need a tool with a GUI interface? Few techniques exist that are both easy to use and professional looking. This article discusses a promising technique that uses the Mozilla platform. It focuses on a rather hard but standard problem: how to display the hierarchical information delivered by the ps(1) command usefully. A recent version of Mozilla (at least 1.4) is required.

Numerous GUI toolkits are available for Linux, from Xt to Tcl/Tk. Tutorials for these kits usually start with a button example. That's very routine, so let's see it and move on. In Mozilla, GUIs are described using XML syntax. A document named button.xul that specifies a button looks like this:

```
<?xml version="1.0"?>
<window xmlns="http://www.mozilla.org/keymaster/
↪gatekeeper/there.is.only.xul">
  <button label="Press Me"/>
</window>
```

The unmanageably long string, http://www.mozilla.org/keymaster/gatekeeper/ etc..., tells Mozilla this file isn't HTML. It's instead XUL, a GUI description language that is Mozilla-specific and a dialect of XML. Make the button's window appear with this command:

```
mozilla -chrome button.xul
```

This example is simple and not worth dwelling on, although there's a lot going on even for a simple button. A ps(1) display is a far more ambitious goal, so let's leap forward.

Instead of the simple <button> widget, one of Mozilla and XUL's bigger guns is required, the <tree> widget. Some coding also is required and a lot more XML. Here, the focus is on fast development, not on seamless perfection. The coding part comes first.

To begin, ps(1) does the initial data gathering. Listing 1 shows the file psdata.ksh, with mode 777.

Listing 1. A Command-Line Wrapper for ps(1)

```
#!/bin/ksh
export COLUMNS=300
ps h -ew -o '%p,%P,%C,%x,%z,%G,%n,%U,%a' \
  > /tmp/psdata
```

The output holds all the interesting fields, comma-separated with no header line. Mandatory components are PID and PPID; the rest are optional but informative fields, such as COMMAND. That's all traditional Linux requires.

The rest of the coding depends on Mozilla technology. The standard compiled distributions provide at least two executables, mozilla and regxpcom. Here, a binary named xpcshell is used as well. This binary is Mozilla's JavaScript equivalent of the Perl interpreter; it has no GUI support. xpcshell sometimes is a good starting point for development, but it is never essential. To acquire this binary, a full compilation of Mozilla is required. First, check toolchain requirements against www.mozilla.org/build. Next, grab the source by FTP or remote CVS. A major release rather than a nightly release is recommended. Once unpacked, follow standard compilation steps:

```
cd mozilla
./configure --disable-debug
make
make install
```

Debug versions are slow and have messy diagnostics; although harmless, they're avoided here. The build takes an hour-plus to finish and requires up to 1GB of space. The resulting binaries are located in mozilla/dist/bin. They can be run from that directory or from anywhere if the MOZILLA_FIVE_HOME and LD_LIBRARY_PATH environment variables are set and exported to that directory's absolute path. Now all the required binaries and shell scripts are available.

With Perl, the output of ps(1) needs to be sucked up into a coding environment. In this case, that's a JavaScript interpreter. To do this, you need more than language syntax—you need support for I/O. In Perl, support is built into core language functions. By comparison, JavaScript has no I/O functions. In Mozilla,

that I/O support is added using objects. Such objects cannot come from a scripting library, because the core language has no I/O. So, a Perl `use` or `require` doesn't work. There are no back-tick operations either, such as `echo `pwd``. Instead, Mozilla has XPCOM.

XPCOM is an implementation of Microsoft's COM, and it works portably on Linux/UNIX, Windows and Macintosh. It's restricted to a single process at the moment; there's no DCOM. XPCOM/COM is the fastest way to add new functionality to a scripting environment. It hooks up a compiled (say C or C++) object to an object reference in the scripting language. The nearest Perl equivalent is XM, but XPCOM does not require the re-linking that XM demands. Mozilla includes thousands of XPCOM objects by default. XPCOM is not some Java-like virtual machine at work, however. XPCOM objects usually are compiled code that runs efficiently on the bare metal.

It might seem strange to use Microsoft ideas on Linux, but XPCOM is fully open source and occupies a UNIX niche that has long been unaddressed: Linux/UNIX lacks a useful intermediate-sized component model. There have been CORBA and dynamic link libraries in the past, but those things are, respectively, very heavyweight and very lightweight. XPCOM is suited perfectly to middle-sized jobs, to application development of large binaries and to performance-critical work. Here it's simply extremely handy.

Use of XPCOM or COM typically includes many calls to the Windows QueryInterface() method. For the sake of Linux programmer sensibilities, this article uses createInstance() and getService() instead. QueryInterface() is available too.

Back to the code. Let's suck up the output of the ps(1) wrapper. Listing 2 shows how.

Listing 2. Batch Loading of Foreign Data into xpcshell

```
const Cc = Components.classes;
const Ci = Components.interfaces;

var klass = {};
var psdata = null;      // last results from ps(1).

klass.file = Cc["@mozilla.org/file/local;1"];
klass.process = Cc["@mozilla.org/process/util;1"];
klass.stream
 = Cc["@mozilla.org/network/file-input-stream;1"];
klass.jsstream
 = Cc["@mozilla.org/scriptableinputstream;1"];

function execute_ps() {
  // freeze until ps(1) is finished.
  var blocking = true, argv = [], result = {};
  var path = "/home/nrm/writing/psviewer/psdata.ksh"

  var file
```

```
     = klass.file.createInstance(Ci.nsILocalFile);
    var process
     = klass.process.createInstance(Ci.nsIProcess);

    file.initWithPath(path);
    process.init(file);
    process.run(blocking, argv,argv.length, result);
  }

  function read_raw_data() {
    const path = "/tmp/psdata";
    var mode_mask = 0x01, perm_mask = 0;  // open(2)

    var file
     = klass.file.createInstance(Ci.nsILocalFile);
    file.initWithPath(path);

    var stream = klass.stream.createInstance(
                             Ci.nsIFileInputStream);
    stream.init(file, mode_mask, perm_mask, 0);

    var jsstream = klass.jsstream.createInstance(
                         Ci.nsIScriptableInputStream);
    jsstream.init(stream);

    var data = jsstream.read(file.fileSize);

    // got the file content. break it down.

    data = data.split("\012");

    for (var i = 0; i < data.length; i++)
    {
      data[i] = data[i].replace(/\s*,\s*/,",");
      data[i] = data[i].replace(/^\s*/,"");
      psdata.push(data[i].split(","));
    }
  }

  execute_ps();
  read_raw_data();
```

The first part of this listing sets up some globals. The Components object is a pre-existing object that acts as a directory of all existing XPCOM objects, called components, and their supported interfaces (in the Java or COM sense). To get an XPCOM object, find the right component (named with a string called a Contract ID) and construct an interface object for it (also named by a string or by a property name; the latter is used here). It's common to reuse components, so once found, they're saved as handy property values on the klass object—class is a reserved word in JavaScript.

Two defined functions are run at the end of this listing. execute_ps() simply executes another process: the ps(1) wrapper script. For that it needs a file object (an nsILocalFile) and a process object (an nsIProcess). run() invokes the process using fork(). Mozilla is designed to do all this portably, but here only Linux is supported because the path of the executable is hard coded as a constant. The other function, read_raw_data(), sucks up the data. Mozilla uses stream, transport and channel concepts the same as do some high-level features of Java, but without the complexity of having to write any classes. A file object is needed for the data file dumped by ps(1). A stream object opens a content pathway to that file. A minor hack is required as well: a special

scriptable stream object must wrap the basic stream. With one read() call the whole file is slurped up into a string. Next, some Perl-like regular expression wizardry breaks the content down into an array of lines and then further into an array of arrays. All data is treated as string data. To see if the data is processed correctly, try using the diagnostic and rudimentary print() method supplied with xpcshell. Alas, Mozilla currently does not support retrieving PIDs, so files named /tmp/psdata.$$ don't work yet. That support is nearly here, though.

Many XPCOM objects are in this script, so how are you to find the right ones? As with any programming library, there's reference material. Look for .IDL files in the Mozilla source code (or under mozilla/dist/idl), on the Web or read a book.

That's enough scripting to start with; scripting and tabular data are well understood in Linux. To build the GUI, Mozilla requires XML, specifically, XUL. That's a different world from the command line, and you have to be familiar with XUL to succeed. Here, the process is broken down into easy stages. First, Listing 3 and Figure 1 show an XUL <tree> widget.

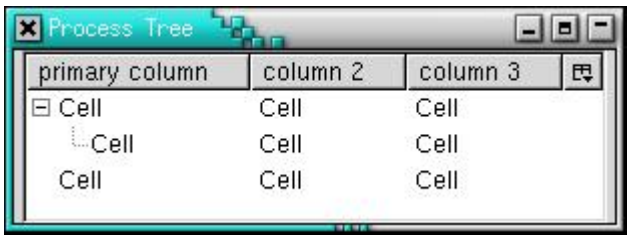


Figure 1. Simple <tree> Widget with Static XUL Content

Listing 3. Plain XUL Code for a <tree> Widget with Static Content

```
<?xml version="1.0"?>
<?xml-stylesheet
  href="chrome://global/skin/global.css"
  type="text/css"?>
<!DOCTYPE window>
<window xmlns="http://www.mozilla.org/keymaster/
↪gatekeeper/there.is.only.xul"
  title="Process Tree">

<tree id="t1" flex="1">
  <treecols>
    <treecol flex="1" id="A"
      label="primary column" primary="true"/>
    <treecol flex="1" id="B" label="column 2"/>
    <treecol flex="1" id="B" label="column 3"/>
  </treecols>

  <treechildren id="titems" flex="1">

    <treeitem id="row1" container="true"
      open="true">
      <treerow>
        <treecell label="Cell"/>
        <treecell label="Cell"/>
        <treecell label="Cell"/>
      </treerow>

      <treechildren>
```

```
          <treeitem>
            <treerow>
              <treecell label="Cell"/>
              <treecell label="Cell"/>
              <treecell label="Cell"/>
            </treerow>
          </treeitem>
        </treechildren>
      </treeitem>

      <treeitem>
        <treerow>
          <treecell label="Cell"/>
          <treecell label="Cell"/>
          <treecell label="Cell"/>
        </treerow>
      </treeitem>

   </treechildren>
  </tree>

  </window>
```

The tree looks nice because the <?xml-stylesheet?> processing instruction drags in the current Mozilla theme for free. Display this tree with the normal Mozilla executable, using the -chrome option to rip away the normal navigation buttons and other decorations:

```
mozilla -chrome static_tree.xul
```

The XML content (henceforth, the code) is a bit like an HTML <table> tag: both column headers and rows of data are specified. The <treeitem> tag is the tricky part; it can contain a <treechildren> tag, which allows the tree to have subtrees, rather than only depth 1 leaf nodes. As seen in Figure 1, the tree widget has a number of interactive features; subtrees can be opened and closed in the same manner as any file explorer application, including Nautilus or Windows Explorer. Columns can be added or deleted using the column picker, the small icon at the extreme right of the tree header that holds column names.

If we wanted, JavaScript scripts could be used to insert the ps(1) data into this XUL document dynamically. That's not hard, and all of the W3C's DOM interfaces are available to do the job. Start by adding Element objects or even use the .innerHTML property. This is an ambitious article, so instead you see a fully data-driven approach, one that avoids hand-constructing any tree.

Listing 4 and Figure 2 show an XUL GUI without a tree. This one has a <template> tag instead

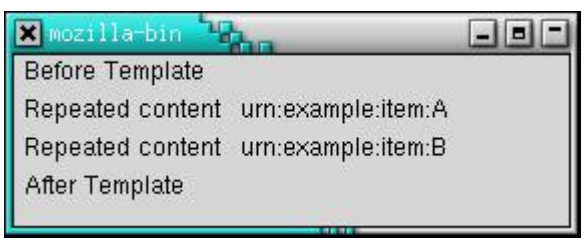


Figure 2. Simple Templated GUI Based on Static RDF Content

Listing 4. XUL Code for Simple <template> Based Content

```
<?xml version="1.0"?>
<?xml-stylesheet
  href="chrome://global/skin/"
  type="text/css"?>
<!DOCTYPE window>
<window xmlns="http://www.mozilla.org/keymaster/
↪gatekeeper/there.is.only.xul">
 <description value="Before Template"/>
 <vbox
  datasources="trivial.rdf"
  ref="urn:example:items"
 containment="http://www.example.org/TestData#items"
>
  <template>
   <rule>
    <conditions>
     <content uri="?uri"/>
     <member container="?uri" child="?note"/>
    </conditions>
    <action>
     <hbox uri="?note">
      <description value="Repeated content"/>
      <description value="?note"/>
     </hbox>
    </action>
   </rule>
  </template>
 </vbox>
 <description value="After Template"/>
</window>
```

An XUL template is like a report template and not like a C++ template. It's the basis for repeated sets of data. The template starts with the <vbox> tag that has a datasources= attribute. The <action> part of the <template> is the content to be repeated for every record that the <conditions> part identifies in the trivial.rdf file. If you're an intermediate at make(1) or SQL or have touched Lisp, Scheme or Prolog, you should be able to grasp how the template system works. Listing 5 shows the trivial.rdf file that drives the display of Figure 2.

Listing 5. Trivial RDF File Matching Listing 4 Template

```
<?xml version="1.0"?>
<RDF

 xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
>
  <Description about="urn:example:root">
    <T:items>
      <Seq about="urn:example:items">
        <li resource="urn:example:item:A"/>
        <li resource="urn:example:item:B"/>
      </Seq>
    </T:items>

  </Description>
</RDF>
```

If this file is modified, Figure 2 can change even though Listing 4 hasn't been altered. That's a data-driven arrangement. This file is RDF, one of the harder W3C standards. Basically, it's a graph of nodes, each node holding three items

of data. The items are called subject, predicate (or property) and object. Simple graphs are trees, so Listing 5 is a tree. Combine the <hbox> in Listing 4 with the <li> tags in Listing 5, and the result appears as illustrated in Figure 2. This is somewhat like an SQL join or join(1). For now, notice that the ref= attribute in Listing 4 matches the <Seq> tag in Listing 5. This is how the two are matched up in Mozilla's template processing logic. Mozilla support for RDF is basic rather than strict, so nearly all the URIs and URLs can be made up on the spot, as though they were variables or constants. That's done throughout this article. Try adding another <li> tag to Listing 5; restart Mozilla and display the page again.

A tree is a good way to display a hierarchical list of processes, and a <template> is a good way to drive the appearance of a tree direct from data. No RDF document is available to work with, though; instead, we have a JavaScript array of records. The solution is to put a <tree> and a <template> tag together and set the RDF file to `rdf:null` = no file. A script is used to create the RDF content directly from JavaScript data. Because of RDF's peculiar design, the content can be dumped into the template in a careless manner and everything simply works. That's a far cleaner but admittedly a more subtle solution than hand-building an XUL tree from JavaScript. Another clean aspect of RDF and templates is the tree can be updated anytime in a simple manner. This means the window can display ps(1) data dynamically, as though a GUI version of `watch ps H` were run. That dynamic step is beyond this article's scope, however.

If the <tree> and <template> tags are put together, the final XUL document is as shown in Figure 3 and Listing 6.

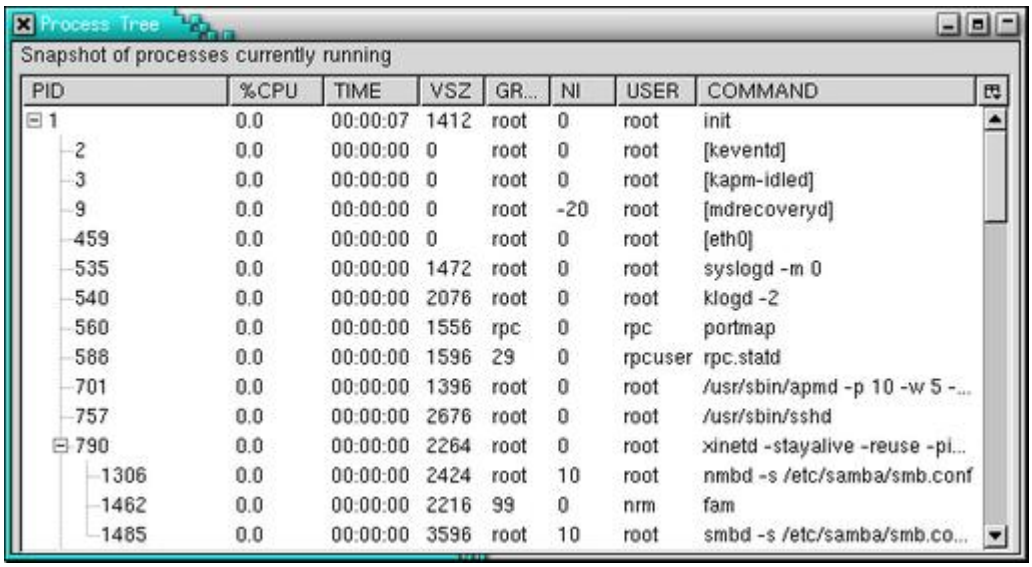


Figure 3. Final <tree> Widget with RDF Data Supplied from JavaScript

Listing 6. Final XUL for a Tree-Based View of ps Data

```
<?xml version="1.0"?>
<?xml-stylesheet
 href="chrome://global/skin/global.css"
 type="text/css"?>
<!DOCTYPE window>
<window xmlns="http://www.mozilla.org/keymaster/
↪gatekeeper/there.is.only.xul"
 title="Process Tree" flex="1">
 <script src="tree.js"/>

 <vbox flex="1">
  <description>
   Snapshot of processes currently running
  </description>

 <tree id="proc-tree"
  flex="1"
  datasources="rdf:null"
  ref="http://www.example.org/ProcData#ProcList"
 containment="http://www.example.org/ProcData#child"
 >
  <treecols>
   <treecol id="pid" primary="true" label="PID"
     minwidth="75"/>
   <splitter class="tree-splitter"/>
   <treecol id="pcpu" label="%CPU" minwidth="40"/>
   <splitter class="tree-splitter"/>
   <treecol id="time" label="TIME" minwidth="40"/>
   <splitter class="tree-splitter"/>
   <treecol id="vsz" label="VSZ" minwidth="40"/>
   <splitter class="tree-splitter"/>
   <treecol id="group" label="GROUP" minwidth="40"/>
   <splitter class="tree-splitter"/>
   <treecol id="nice" label="NI" minwidth="40"/>
   <splitter class="tree-splitter"/>
   <treecol id="user" label="USER" minwidth="40"/>
   <splitter class="tree-splitter"/>
   <treecol flex="1" id="args" label="COMMAND"
     minwidth="40"/>
  </treecols>
  <template>
   <treechildren>
    <treeitem open="true" uri="rdf:*">
     <treerow>
      <treecell
 label="rdf:http://www.example.org/ProcData#pid"/>
      <treecell
 label="rdf:http://www.example.org/ProcData#pcpu"/>
      <treecell
 label="rdf:http://www.example.org/ProcData#time"/>
      <treecell
 label="rdf:http://www.example.org/ProcData#vsz"/>
      <treecell
 label="rdf:http://www.example.org/ProcData#group"/>
      <treecell
 label="rdf:http://www.example.org/ProcData#nice"/>
      <treecell
 label="rdf:http://www.example.org/ProcData#user"/>
      <treecell
 label="rdf:http://www.example.org/ProcData#args"/>
     </treerow>
    </treeitem>
   </treechildren>
  </template>
 </tree>
</vbox>

</window>
```

Again, you can spot the datasource= and ref= attributes and the <template> tag. The URLs beginning with rdf: indicate spots where RDF data should be put into the template. In the earlier example, variables started with a question

mark. Two syntaxes are available to mark such spots. Not surprisingly, there's one such piece of data for every column and every row.

The <splitter> tag is simply friendly decoration; it allows the user to resize the columns. Doing so aids readability, as do the minwidth= and flex= attributes. Figure 3 shows how the displayed process hierarchy naturally fills the tree.

Near the top of Listing 6, a <script> tag includes all the code from Listing 2, plus more. When such scripts are included, there is an immediate security problem. The problem is Mozilla technology must ensure secure display of remotely located files and scripts, such as HTML pages. This is like the Java Server of Origin rule. xpcshell is entirely unsecured, but the main Mozilla binary has normal security. With an intensive configuration effort, security restrictions can be overcome, but it's simpler to register the script as a package. To do that, all the files have to be moved to the chrome, a directory inside the Mozilla install area where all security restrictions are lifted. How to do that is explained shortly, but first we finish the application with a script that moves the ps(1) data from a plain JavaScript data structure into an RDF datasource. This script replaces the static RDF file used earlier (Listing 7).

Listing 7. Insertion of Facts into an RDF Datasource from a Script

```
// --- globals ---
klass.datasource
 = Cc["@mozilla.org/rdf/datasource;1" +
      "?name=in-memory-datasource"];
klass.rdf
 = Cc["@mozilla.org/rdf/rdf-service;1"];

var schema = "http://www.example.org/ProcData#";
var props =
  [ "pid", "ppid", "pcpu", "time", "vsz",
    "group", "nice", "user", "args" ];

var rdf = klass.rdf.getService(Ci.nsIRDFService);

var root  = rdf.GetResource(schema + "ProcList");
var child = rdf.GetResource(schema + "child");
var preds = [];

for (var p in props)
  preds[p] = rdf.GetResource(schema + props[p]);

// --- mainline ---

window.addEventListener("load",load_handler,true);

// --- functions ---

function update_tree() {
  var tree = document.getElementById("proc-tree");

  // get the in-memory ds, not the rdf:localstore
  var ds   = tree.database.GetDataSources();
      ds   = ( ds.getNext(), ds.getNext() );
      ds   = ds.QueryInterface(Ci.nsIRDFDataSource);

  var sub, pred, obj;

  for (var i=0; i < psdata.length; i++)
```

```
    {
      if ( psdata[i][1] == "0" )  // a root node
        sub = root;
      else                        // a child node
        sub = rdf.GetResource(
                schema + "process-" + psdata[i][1]);

      pred = child;
      obj = rdf.GetResource(
                schema + "process-" + psdata[i][0]);

      ds.Assert(sub, pred, obj, true);

      // add all properties for this process

      sub = obj;
      for (var j=0; j < psdata[i].length; j++)
      {
        pred = preds[j];
        obj = rdf.GetLiteral(psdata[i][j]);
        ds.Assert(sub, pred, obj, true);
      }
    }
  }

  function load_handler() {
    var tree = document.getElementById("proc-tree");
    var ds = klass.datasource.createInstance(
                       Ci.nsIRDFInMemoryDataSource);
    tree.database.AddDataSource(ds);

    update_tree();
  }
```

Listing 7 shows the extra script logic that substitutes for a static RDF file. Adding the JavaScript data to the RDF used by the tree's template requires a process of steps. Mozilla sucks up RDF data into an object called a datasource. Because rdf:null has been specified, no datasource object exists, so one must be created and attached to the template. load_handler() does that, after the document is loaded safely. Using an onload handler is a standard HTML trick, and such tricks apply equally well to XUL. The update_tree() function then fills that datasource with RDF content for the template. It's done pretty simply. A double loop steps through each data item in the JavaScript array. For each ps(1) process, Assert() is called to create one RDF node of data (a triple of three items) that states PPID X has child PID Y and a further set of RDF nodes that states PID X has USER A or PID X has GROUP B. The <template> and the <tree> tag work together to sort those RDF nodes automatically into a tree arrangement; this is like make(1) calculating the dependency tree for all the targets stated in a given Makefile. With this script acting in place of a static RDF file, the simple process viewer is complete. Finally, the steps required to lift security by registering the code as a package are:

```
M5H = $MOZILLA_FIVE_HOME
mkdir -p $M5H/chrome/psviewer/content
cp *  $M5H/chrome/psviewer/content
vi    $M5H/chrome/psviewer/content/contents.rdf
vi    $M5H/chrome/installed-chrome.txt
```

Listing 8. Chrome Registration Information for the psviewer Package

```
<?xml version="1.0"?>
<RDF
xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
>

 <Seq about="urn:mozilla:package:root">
  <li resource="urn:mozilla:package:psviewer"/>
 </Seq>

 <Description about="urn:mozilla:package:psviewer"
   chrome:displayName="PSViewer"
   chrome:author="Nigel McFarlane"
   chrome:name="psviewer">
 </Description>
</RDF>
```

The first vi editing session creates the file contents.rdf. It must look exactly like Listing 8. The second vi editing session adds to the file installed-chrome.txt. A single line is added:

```
content,install,url,resource:/chrome/psviewer/content/
```

When Mozilla starts up, it examines this last file. If it is modified, the directories listed are examined for contents.rdf files. Those files are then read, and like make(1), Mozilla builds in its head a picture of all the packages known to exist. All known packages have full security access, and Listing 8 adds the package psviewer. The secure files now can be displayed and run with a URL such as:

```
mozilla -chrome chrome://psviewer/content/tree.xul
```

instead of:

```
mozilla -chrome file:///home/nrm/psviewer/tree.xul
```

The psviewer tool has first-class status within the Mozilla installation. If necessary, it could be integrated with other applications, such as the Firefox/ Firebird browser or Thunderbird e-mail client. It also could be added as a menu option to the Tools menu, for example.

There's a lot of technology in this article. The biggest mistake you can make is to try to use all the features described here in your first Mozilla experiment. Because validation of XML is less than verbose in Mozilla, you easily can become tied in a knot. It's best to start with a simple project and work up to the challenging combinations played with here. Although the output of ps(1) also can be made into a dynamic HTML page, XUL is a more robust and professional GUI in the end, fully integrated with the desktop.

Mozilla is a powerful GUI environment waiting to be explored. It is likely to occupy the same niche under Linux that Visual Basic occupies under Windows. Even better, Mozilla is a portable and cross-platform technology. Your projects

can be designed to work on BSD, HP-UX, SunOS, AIX and Mac OS X, as well as Linux.

Nigel McFarlane is a freelance science and technology writer with an extensive programming background. His latest book is *Rapid Application Development with Mozilla*, ISBN 0131423436. Reach him at nrm@kingtide.com.au.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Eclipse Goes Native

John Healy

Andrew Haley

Tom Tromey

Issue #123, July 2004

Red Hat's Eclipse Engineering team has freed the popular integrated development environment from its ties to a proprietary Java Virtual Machine.

Eclipse is an open-source, extensible integrated development environment (IDE) that's growing quickly in popularity. Written in Java, it provides a multilanguage development environment that allows developers to code in Java, C and C++. In response to the need for improved performance and additional platform coverage for the Red Hat Developer Suite, of which Eclipse is the core, we created a version of Eclipse that's compiled natively. Instead of running on top of a virtual machine the way Java programs usually do—although that can still be done if the user prefers—Red Hat's version of Eclipse is compiled to binary and runs natively using the libgcj runtime libraries, similar to the way a C program runs using the GNU C libraries.

To compile Eclipse natively, Red Hat's Eclipse Engineering team used GCJ, a free, optimizing, ahead-of-time compiler for Java. GCJ can compile Java source code to native machine code, Java source code to Java bytecode and Java bytecode to native machine code. The approach we took involves using GCJ to compile Java bytecode to native machine code.

This article discusses why native compilation was an attractive choice; explains what we had to do to GCJ, libgcj and Eclipse to make it possible; and shows, using a real-world example, that open-source Java has come a long way and now is useful commercially.

Motivation

Two main factors from the early days of Developer Suite planning and engineering drove us toward native compilation: platform coverage and performance. Red Hat Enterprise Linux was scheduled to ship on several 64-bit architectures, and we wanted to make sure Developer Suite could run on all of them. One big problem was Eclipse had never been run on a 64-bit platform and it contained some code, specifically the interface between SWT, the graphics toolkit in Eclipse, and its native C libraries, that assumed 32-bit addresses. Aside from having to create a clean 64-bit version of SWT, we were faced with a more significant problem: no 64-bit Java Virtual Machine (JVM) for x86_64, AMD's 64-bit architecture, existed at the time, and it didn't look hopeful that one would be available before we had to ship.

Another problem we had was performance. Eclipse worked well on Microsoft Windows but the version available at the time was pretty slow on Linux. We found that startup alone took well over a minute, and early user testing found that the interface was a little too sluggish for comfortable use. For example, Eclipse is based on perspectives, which are collections of views and editors, only one of which is visible at a time. Switching between them is something that a user does fairly frequently. However, changing perspectives introduced substantial delays we thought unacceptable for the enterprise development market Red Hat Developer Suite was targeting.

The solution we came up with was to use GCJ to compile Eclipse into native binaries that could run without having a JVM installed. We knew that native compilation would help with the performance problems, because we would no longer have the overhead that comes with the JVM layer. It also would solve the platform coverage problem, as GCJ/libgcj was available on all of the 64-bit platforms we had to support, although in some cases, such as x86_64, it still needed a lot of work. Native compilation solved the technical problems we had and gave us the additional benefits of reducing our external dependencies, allowing us to make some significant improvements to open-source Java and to demonstrate that open-source Java has matured to the point of being useful commercially.

Approach

At the outset of this project, we really didn't know if it was possible to compile Eclipse with GCJ and expect it to run. First, Eclipse is a large program—more than two million lines of code as counted by `wc`. We didn't know much about Eclipse internals or what runtime facilities it might use. Second, GCJ's background is in embedded systems, and we knew that work remained on parts of the Java programming language, class loaders in particular, which are used heavily by Eclipse. Third, the free class libraries were not complete. We

didn't know if Eclipse could use facilities we hadn't written yet or even whether Eclipse might break the rules and use internal, undocumented com.sun.* interfaces, as too many Java programs seem to do.

We therefore took a two-pronged approach to determining whether a project like this could succeed. First, we used GCJ to make a list of the APIs used by Eclipse that we did not or could not implement. To accomplish this, we wrote a shell script that would try to compile each Eclipse Java archive library (jar file) to object code. We then looked through the error messages to see what was missing. The results of this script were not encouraging: we found a large number of missing packages. Still, more investigation was required because some things didn't make sense. For instance, there were dependencies on the Swing graphical user interface classes, but we knew that Eclipse used SWT and not Swing.

Further investigation showed that many of the weird undefined references came not from Eclipse itself but from the third-party jar files included with it. For example, Eclipse includes its own copy of the Ant build tool and its own copy of the Apache Tomcat dynamic Web server. We knew that in many cases, the referenced classes would not actually be invoked in the Eclipse environment. This encouraged us to take another look at how to get Eclipse working.

Our second angle of attack was to try running Eclipse using the bytecode interpreter that comes with libgcj. By doing this, we reasoned, we would concentrate on runtime bugs, including the aforementioned class loader problems and missing functionality actually used by Eclipse.

This approach also was discouraging initially. We ran into problems not only with class loading, but also with the fact that libgcj's implementation of protection domains needed work. These are the bases for Java's secure sandbox architecture, which allows untrusted code to be run in a secure way. Problems in this area had an unfortunate shadowing effect—we had to fix each bug before we could discover the next one.

Changes to libgcj

Our first round of changes to libgcj was bug fixing only. We implemented protection domains properly. Then, we made a pass over the entire runtime, fixing bugs related to class loading. Because of the way class loading had been implemented in libgcj, we had to modify all the places in the native code that conceivably might load a class to forward the request to the appropriate class loader.

Once this was done, we were able to start Eclipse using the libgcj bytecode interpreter. At this point the question became, how can we take real advantage of GCJ to compile Eclipse?

The naïve approach to this dilemma, namely precompiling all the classes and linking them all together, had been ruled out by our investigations into Eclipse's internals. This approach would clash with Eclipse's relatively sophisticated class loading strategy.

More investigation revealed that most classes are loaded by instances of the DelegatingURLClassLoader, which is a subclass of the standard URLClassLoader that has been extended to understand Eclipse's plugin architecture. It seemed like the best approach was to modify Eclipse to allow it to load precompiled shared libraries as well as bytecode files. We reasoned that the required changes would be localized due to the way plugin class loading had been structured.

In fact, we had to go one step further and extend libgcj a bit as well. libgcj knew how to load shared libraries invisibly in response to a call to, for example, Class.forName(). However, this magic always happened at the level of the bootstrap class loader. That wouldn't work well for Eclipse or for any other application that defines its own class loaders, so we invented a new gcjlib URL type. This is like a jar URL, but it points to a shared library. We also made some minor extensions to our implementation of URLClassLoader so that gcjlib URLs would be treated specially.

Doing this wasn't enough, however. We also had to solve the linkage problems. In particular, if we compiled a jar file to a shared library, how could we prevent the dlopen() of such a shared library from immediately failing due to unresolved symbols? The solution to this problem was to resurrect and clean up the -fno-assume-compiled option in GCJ. This option, which never had been finished, enabled an alternative ABI that caused GCJ's output to resolve most references at runtime rather than at link time.

The -f-no-assume-compiled option has various limitations and inefficiencies. On the boards for the future is a cleaner way to achieve this same goal. On the GCJ mailing list (see the on-line Resources section) this option is referred to either as the binary compatibility ABI or -findirect-dispatch. This new ABI does everything -fno-assume-compiled does, but in a much more efficient and compatible way. Development is underway and is coming along nicely on this new feature, one of several contributing to GCJ's enterprise readiness.

Changes to Eclipse

Once all this was in place, we finally were ready to make our changes to Eclipse. These turned out to be remarkably small. Most of the work involved making the same sort of change in three different places. In essence, we modified Eclipse so that when it's looking for a plugin's jar file, it also looks for a similarly named shared library installed alongside it. If there is one, we rewrite the URL passed to the class loader from a jar URL to a gcjlib URL. All rewriting is done conditionally, so our natively compiled Eclipse still works with an unmodified JVM. In other words, users are not locked in to native compilation if they would rather use a JVM instead.

Once that was done, we wrote our own launcher that understood how to bootstrap the Eclipse platform from shared libraries. This was accomplished in a modest 90 lines of code.

Profiling

After all that, Eclipse was mysteriously slow. Had we done something wrong? Was GCJ-compiled code substantially worse than the code generated on the fly by the current crop of just-in-time (JIT) compilers? Did -fno-assume-compiled have enormous overhead?

One nice advantage of GCJ is its output generally can be treated in the same way one treats any object code. That is, existing tools such as OProfile can be applied to it directly without any change. And that, in fact, is how we investigated our performance problem.

The first thing we noticed was a large number of exceptions being thrown during platform startup. Amid the grumblings of compiler writers (exceptions should be for *exceptional* circumstances), and although we were considering changes to the GCJ runtime that would violate Java semantics, we noticed a strange symbol in the OProfile output. It turned out that a small bit of buggy assembly code deep in the libgcj runtime was causing a linear search of exception handling tables rather than the expected binary search. The overhead of this search through the entire program every time an exception was thrown was vast. A fix to the errant assembly code proved this was the problem, and suddenly our natively compiled Eclipse was able to start a second faster than the stock version using a JVM. To quantify it a bit further, the startup time dropped from more than a minute before the fix to less than 15 seconds after it.

Limitations and Shameless Hacks

Currently, we don't compile Eclipse directly from source to object code. Instead, we compile to bytecode and then compile the jar files to shared libraries. This is done for two reasons. First, a few bugs in the GCJ source compiler haven't been fixed. Second, Eclipse comes with its own build scripts that compile from source to bytecode. Reworking the Eclipse build system to allow building directly from source to binary seemed like a much larger divergence from the upstream sources than we were willing to maintain.

Also, we currently don't precompile all the jar files to shared libraries—some remain as jar files and are interpreted at runtime. This is done because the class libraries still are incomplete, and these jar files refer to classes that have not been implemented yet.

One of our patches is unsuitable for the public GCJ. We had to disable the compile-time bytecode verifier, as it was too buggy to compile some of the Eclipse jar files. We're in the process of replacing this verifier with a more robust one.

In addition, one limitation of natively compiled Eclipse deserves mention. You can't use natively compiled Eclipse to debug a GCJ-compiled application, because JDWP, the Java Debug Wire Protocol used by Eclipse, hasn't been implemented in libgcj yet.

Implications and Future Directions

The achievement of the native compilation of Eclipse is a strong indication that open-source Java based on GCJ and libgcj/classpath has reached the point of being commercially useful. That said, it's still not complete. Some fairly substantial gaps still need to be filled in before open-source Java can be a proper drop-in substitute for proprietary JVMs.

One of the major areas that needs work is the development/integration of a JIT compiler. JIT would allow a GCJ-based open-source Java environment to be used in a manner similar to a conventional JVM, meaning that native compilation and platform-specific binaries would not be necessary for performance reasons.

The other major piece that needs work also is, by far, the most visible missing piece—Swing. Work on an open-source implementation of Swing is coming along nicely as part of the GNU Classpath Project, but Swing is a huge undertaking and the GNU Classpath implementation is still not quite usable.

A full-featured and completely open-source Java environment is an attractive alternative to proprietary JVMs, and it's now within reach. During the past six

months, Red Hat has more than doubled the number of engineers working in support of the Open Source Java solution and community. Eclipse is a large, complicated piece of software, and natively compiling and running it was an excellent test of and testament to the progress being made on open-source Java. The power of open source lies in its communities, so please consider joining the open-source Java community and contributing to the GCJ and GNU Classpath Projects in any way that interests you.

Resources for this article: /article/7549.

John Healy is the manager of Red Hat's Eclipse Engineering group, based in Toronto (people.redhat.com/jhealy). In the past he's worked on custom open-source toolchains for embedded processors as well as CRM and computer-telephony applications.

Andrew Haley has been a programmer for longer than he cares to remember. He is one of the maintainers of GCJ. He works for Red Hat, which supports him in this task.

Tom Tromey has worked on free software since the early 1990s. Patches of his appear in GCC, Emacs, GNOME, Autoconf, GDB and probably other packages he has forgotten about. He works at Red Hat as the technical lead of the Eclipse Engineering team. He can be reached at tromey@redhat.com.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Clusters for Nothing and Nodes for Free

Alexander Perry

Hoke Trammell

David Haynes

Issue #123, July 2004

When the users are away, your company's legacy desktop systems can become a powerful temporary Linux cluster.

At Quantum Magnetics we do contract R&D. We often need to design silicon chips, simulate electromagnetic systems and analyze masses of data from field tests. When a single set of regression tests started taking longer than a working day to perform, coauthor Alex Perry found himself wondering how to get short-term access to a cluster. We describe here the sequence of steps that enabled us to set up an OpenMosix cluster with little effort and without having to purchase anything.

Each productivity increase justified putting time into the next step of bringing up the company-wide cluster. We omit details here that are provided in the instructions and FAQs for each project (see the on-line Resources section), partly because things will have changed by the time the article goes to print and partly for brevity.

Choose an Application

The simplest applications to run on a cluster are command-line based and run as multiple instances on one computer. Applications don't have to be written specifically for Linux, because they could use WINE or another portability layer. If multiple instances are not possible, much more time has to be put into providing a virtual machine abstraction layer. It is worth checking your specific application before putting any effort into building a cluster to see whether it is capable of benefitting from an OpenMosix-based cluster.

Most of our logic code is written in Verilog partly because, as the joke goes, we can't type fast enough to use VHDL. Mainly, though, our reason is that a broader range of tools is available in Verilog. We use several closed-source place-and-route tools under Microsoft Windows, the runtime of which is tiny, so putting these on the cluster is not worth the effort. For simulation, we have both open- and closed-source options. It is convenient to use the graphical tools (all closed-source, unfortunately) that have IDE source-level debuggers when trying to track down a bug, but these either don't like clusters or have a hefty licensing price tag when running on a cluster. We use Icarus Verilog for non-interactive simulations, as regression testing is more than 99% of the total simulation workload. We like it because multiple simulators can run in parallel; each simulator is a single Linux process; the tool has its own public regression suite; the developers are helpful and responsive; and the syntax parser is paranoid and accurate.

The paranoia of the syntax parser flags a lot of problems for us. Many parsers simply select one interpretation of ambiguously written source, leading to incorrect behaviour that is effectively a bug. In contrast, Icarus immediately complains about ambiguities, and after we've made the tiny rewrite, the synthesized chip suddenly starts working the way that it was intended.

The developers for Icarus, by responding rapidly to bug reports and patches, enhance the value of the simulator in our work. We update from CVS to benefit from those almost-immediate source changes. In addition, it is much easier to standardize one virtual machine (the cluster) than to manage the versions on the individual workstations.

We run all our proprietary simulation tests immediately before and after a new version of Icarus is retrieved from CVS. About once a year, the simulation results are different, so we submit a bug report that localizes the problem to a test case outside our proprietary work. In this way, all our proprietary work acts as an additional regression suite for the Icarus Project without us having to make it available to our competitors. It also ensures that any official release of Icarus is useful to us.

Listing 1. Extract from a chip design project Makefile showing how regression happens. The %.txt reference data is generated automatically using C and AWK.

```
TESTS = test1 test2 test3 test4
.PHONY:default
default: $(TESTS:%=%.log)
        -hostname -f
        -cat $(TESTS:%=%.log)

%: %.vl
        iverilog -o$* $<
%.vl: %.v
        ./Makefile.sh $^ > $@
```

```
test1.vl: source5.v source7.v

%.log: Makefile %.txt %.out
        ls --full-time $*.out > $@
        diff -b -C2 $*.txt $*.out >> $@
        echo "... PASS ..." >> $@

.PRECIOUS: %.out %.txt
%.out: %
        time ./$* > $@
.PHONY: %.batch batch
%.batch: %
        echo "make $*.log" | batch -m
batch%: $(TESTS)
        echo "make -l $*.5" | batch -m
```

The shell script Makefile.sh ensures that Icarus and the Makefile always agree on what source files are being used to build the simulation:

```
#! /bin/bash
echo -n "// "; date
for item in $*; do
    echo "\`include \"$item\""; done
```

In our engineering design work, we use `make`, as shown in Listing 1, to automate test execution and to manage all the Verilog source files, the reference implementation in C, validated test data, the pool of regression tests and all the simulation results.

Without the cluster, between six and ten hours were needed to complete all the dependencies that resulted from a minor change to a source file. Logic simulation usually is about a factor of a million slower than real life, so the regression simulates only about 20 milliseconds of time. The tests have to be selected carefully, because the board can run for as long as 30 seconds per use (about a year of simulation).

Protect History

The most valuable part of the work is the data and all the intermediate states of the work in progress, because any damage here sets you back days even if you have backups and external version control checkpoints. A RAID array of 1 or 5 is the usual protection. One computer, not one of the fastest ones, should have at least two hard drives on distinct controllers. It is worth making sure that each drive has a small swap partition so the kernel can use all the swaps and do some load balancing.

Turn on the kernel-space NFS server and configure /etc/exports from the point of view of securing the data storage from damage. When the NFS is under heavy load, user-space programs have to be swapped to make space for additional disk cache. Consider having a runlevel that could be deferred to disable all the services that wake up periodically for minor purposes.

We're using an existing dual-Athlon MP machine with over a terabyte of storage and running Debian stable as our NFS server. The system is overkill for the cluster; we originally built it to archive field test data and then stream the data to multiple clients for analysis. No X server is used, because the cooling fans make so much noise that nobody wants the machine sitting next to his or her desk.

Without a Cluster

Using `make batch2` on a dual-processor machine reduced our runtime by about 40%, with one of the processors being idle near the end of the run. The total runtime was between four and six hours of clock time. This can be improved, even without a cluster, by distributing the work across many machines using OpenSSH with public key authentication. The *Linux Journal* article ("Eleven SSH Tricks" by Daniel R. Allen, August 2003) explained how to configure this powerful package to avoid endless streams of password prompts while simultaneously enhancing network security.

Listing 2. This runs simulations in parallel on many computers. The runtime is consistent but can be inefficient.

```
#! /bin/bash
for pair in host1/test1 host2/test2 \
            host2/test3 host5/test4
do test=`basename $pair`
   make $test
   ssh `dirname $pair` vvpstdin \
       < $test > $test.out &
done
wait; make
```

The Icarus simulation engine vvp cannot load from standard input, so we use this vvpstdin script:

```
#! /bin/bash
F=/tmp/`basename $0`.tmp.$$
cat > $F
/usr/local/bin/vvp $F
exec rm $F
```

The machines sharing the work usually come to have different performance capabilities. It is important to match the relative runtimes of the various tests against individual processor speeds, remembering SMP, so all of the tests finish at about the same time. We found it best to optimize the mapping manually in a script like the one shown in Listing 2.

By using SSH to two dual-Athlon MP machines, one Pentium III laptop and five Pentium II desktops, we reduced runtime to a fairly consistent two hours.

Initial Cluster

If everyone is running the same version of the same distribution, it probably is sufficient to install the prepackaged binaries of OpenMosix. Thereby, you have the workload migration available without any effort. Always use the autoconfiguration option instead of specifying the list of nodes manually, because the cluster grows in later stages.

We use several different distributions in the office, so we downloaded a pristine 2.4.20 kernel tarball, the matching OpenMosix patch and the source of user-space tools to the NFS fileserver. After making careful notes of the configuration settings to keep all the machines in step, we followed the instructions on the OpenMosix Web site. Because it takes our time and effort to recompile and reinstall kernels, we modified only four computers needed to cluster seven processors. This is slightly less capable than the ten processors achieved through SSH. Even so, the worst-case runtime stayed almost identical, because the migration did the load balancing slightly better than our hand-optimized script could achieve. Because Alex could use `make -j` and let OpenMosix assign the work, all incremental workloads completed faster and did not need the full two hours.

OpenMosix tries to be fair and have all programs run at the same speed by putting more work on the faster computers. This is not optimal for the logic simulation workload, however, as we usually know the relative runtimes. In this case, a short script (not included here) helpfully monitors the contents of /proc. The script periodically looks for process pairs with a big ratio in their expected runtimes but whose node assignments are not providing a corresponding execution speed ratio. The script uses its knowledge of prior runs to request a migration to gain a long-term benefit hidden from OpenMosix. Such a script is not needed if, for your application, the runtimes of all processes are similar.

Those Old Machines

Usually, plenty of spare older computers can be found hiding in corners. Put an X server on one of them that is configured to be a terminal into the xdm service on the fast computers. With this machine, you can shut down the X servers on the fast computers and release their processor and memory resources back into the important workload. Alex's desktop computer, a 400MHz Pentium II, already had its X server indirecting over xdm's chooser. David's work keeps him roaming the building and relying on VNC, so he already was using Xvnc. Only Hoke needed to make minor changes to configuration files.

Next, install LTSP on one computer and set up all the other old computers to use diskless boots to become terminals too. Doing so eliminates the administration of all those operating systems. You now should have enough

terminal stations that all your team members are using terminals, and all the fast compute nodes can stay in the stripped runlevel and be as efficient as possible. It doesn't take long to get those two features working, and an excellent time to work on this is whenever you're waiting on the running jobs.

There is no need to get the DHCP and TFTP components of LTSP working. Put the kernel on a floppy, together with SysLinux configured to trigger the non-boot DHCP, and mount the NFS root filesystem. Then, use that one floppy to do the one-time boot of the terminals. Reboots are needed infrequently, so the slowness of the floppy is fine.

Coworkers

Once the cluster and LTSP are both functional, we simply combine them. The short script shown in Listing 3 uses the NBI tools to put the patched kernel into /ltsp/i386/boot. Our DHCP server's filename parameter is a soft link, so we can change the LTSP kernel rapidly while testing upgrades. After copying the user-space tools into the client filesystem and renaming the init script as rc.openmosix, we add the few lines in Listing 4 to the LTSP startup script. Slower computers have `MOSIX=N` in the LTSP configuration file because they would not contribute much performance to the cluster.

One line in /ltsp/i386/etc/inittab:

```
ca:12345:ctrlaltdel:/sbin/ctrlaltdel
```

calls a copy of Debian's shutdown binary using the script shown in Listing 5. This ensures that Ctrl-Alt-Del forces a clean disconnect from the cluster before rebooting.

Listing 3. This /ltsp/i386/usr/src/netkernels copies kernels from the build tree to the TFTP directory.

```
#! /bin/bash
for vsn in 2.4.20 2.4.21
do pushd linux-$vsn; make bzImage; popd
   s=linux-$vsn/arch/i386/boot/bzImage
   d=../../boot/vmlinuznbi-$vsn
   mknbi-linux --ip=dhcp \
      --append "root=/dev/nfs" $s >$d
done
```

Listing 4. These few lines are appended to the LTSP startup script /ltsp/i386/etc/rc.local.

```
MOSIX=`get_cfg MOSIX Y`
if [ "$MOSIX" = "Y" ]; then
  echo 1 > /proc/hpc/admin/lstay
  AUTODISC=1 /etc/rc.openmosix start
fi
```

Listing 5. New Shutdown Script

```
#! /bin/bash
prefix="Control Alt Del detected: "
echo "$prefix OpenMosix"
/etc/rc.openmosix stop
echo "$prefix ShutDown"
/sbin/shutdown -r -n now
echo "$prefix failed (give up)"
```

Once you are confident that the LTSP-OpenMosix kernel is stable and not going to be changed, you can hand out floppies with the new kernel. The LTSP users won't see a difference, but your compute workload will.

If you would like to maintain the option of changing the kernel without having to hunt around the company to find all the old floppies, now is a good time to get the DHCP network boot working. The LTSP documentation describes how to configure Linux or UNIX servers, but our implementation was running on Microsoft Windows. David, who administers our Windows-based DNS and DHCP servers, set up Netboot in DHCP (Figure 1).

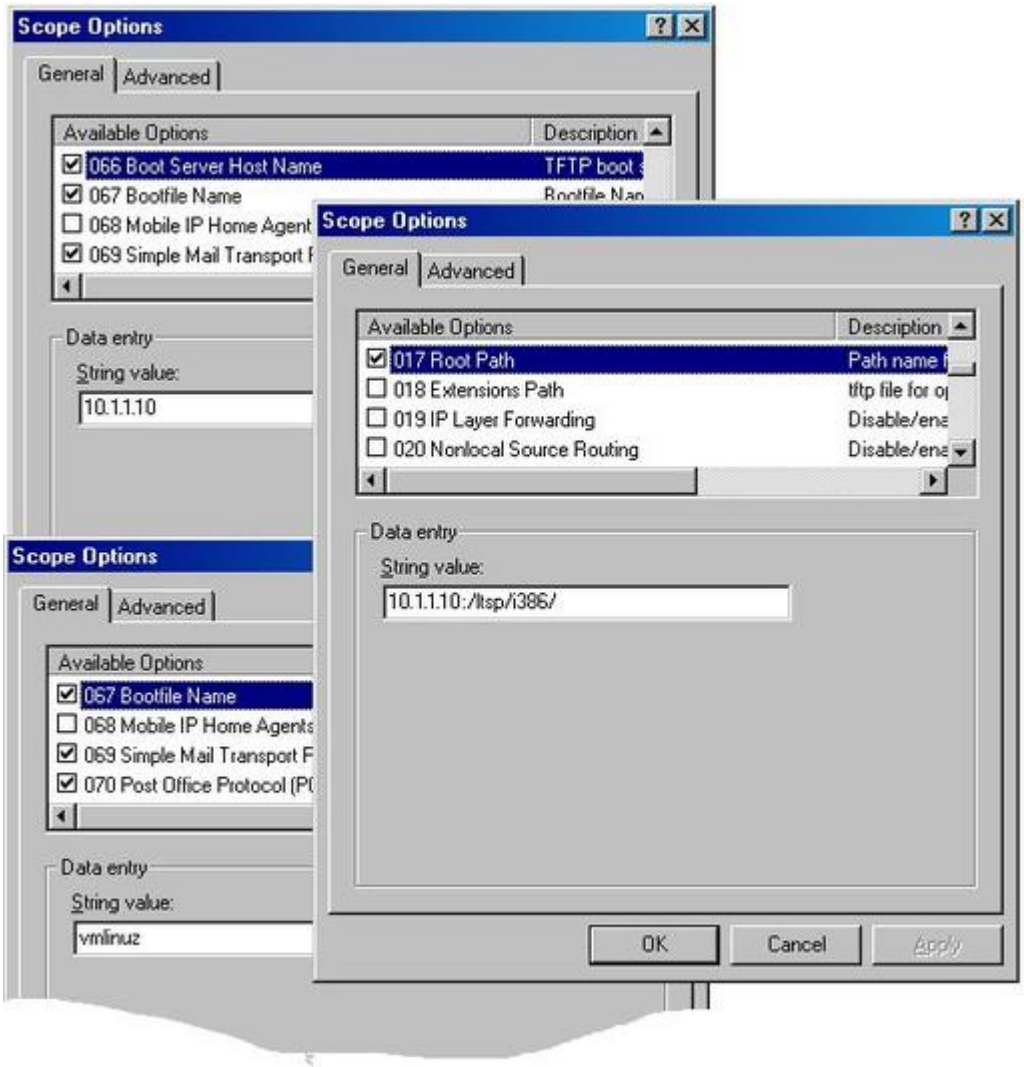


Figure 1. The three scope entries needed on a Windows DHCP server. Notice that the root path has the trailing slash workaround.

Microsoft DHCP appends a null to the NFSROOT, as discussed in LTSP mailing lists, so you need a soft link:

```
ln -s /ltsp/i386 /ltsp/i386/000
```

Listing 6. This /ltsp/i386/usr/src/bootfloppy makes a floppy network boot for several models of network card.

```
#! /bin/bash
if test "$EUID" != "0"; then exec sudo $0; fi

# Configuration options
L="eepro100 rtl8029 rtl8139 tulip 3c905c-tpo"
E=etherboot-5.0.10/src
item=3c905c-t

F=${0}.img
M=$F.mnt
C=$M/syslinux.cfg
CC=$M/toc.txt

# Create the virtual bootable floppy disk
dd if=/dev/zero of=$F bs=1024 count=1440
```

```
mkdir -p $M; mkdosfs $F; syslinux $F
mount -t vfat -o loop $F $M

# Populate the floppy with configuration files
cat <<END >$CC
This floppy image is at http://ltsp$F
The bootloaders are built using $E
If you don't have a $item, you need to type
in the card name below. If your network card is
not listed, please notify $USER@qm.com To change the
default permanently, you need to edit the
file `basename $C`
END

cat <<END > $C
display `basename $CC`
prompt 1
timeout 100
default $item
END

# Now add the bootable images
for item in $L
do      T=bin32/$item.lzlilo
        pushd $E; make $T; popd
        item=${item:0:8}
        cp $E/$T $M/$item
        echo >>$CC "     $item"
done
flip -m $C
flip -m $CC

# Release the floppy disk
df $M; umount $M; rmdir $M
```

For years, our LTSP deployment has been providing multiple X stations to various engineering computers, and we never needed a central application server. The script shown in Listing 6 builds a floppy image for use with all computers. The user simply specifies the network card model.

With this infrastructure, any cluster user can stroll through the buildings with one of those floppies and reboot idle machines into the cluster until sufficient resources are available to run workloads efficiently. For logic simulation, Alex simply adds machines until there are more fast computers in the cluster than slow tests in the suite, so the regression never takes longer than 16 minutes. With that efficiency boost, he rapidly finished the design. Without running mtop, you'd never notice OpenMosix migrating compute-bound processes into the cluster. Meanwhile, others are using the network for different projects.

Large Off-Peak Cluster

Quantum Magnetics has about 100 employees, so our cluster is limited to around 100 nodes, as a few people have more than one computer. We're setting things up so that machines spend nights in the cluster and days as normal user workstations. They reboot at least twice every day and check a configuration directory to decide whether to boot from the network or from the hard drive.

The BIOS must be configured to try the PXE boot before the hard drive. The DHCP servers distinguish between EtherBoot and PXE boot requests, with the latter receiving the boot filename for PXELINUX. There are two directories of configuration files, one for day and one for evening, and a small cron job to switch between them. The daytime boot chains to the master boot record on the hard drive, and the evening boot chains to the PXE version of EtherBoot.

The LTSP configuration file indicates which machines have to reboot on weekday mornings and causes the ctrlaltdel script to run. If a user comes to work early, simply pressing Ctrl-Alt-Del brings the machine back into daytime mode as soon as possible.

Remote Windows administration is used to force workstations to log off after inactivity in the evening and then reboot once. If either of the two network boot stages fail, the machine starts Windows and does not join the cluster.

Long-Term Use

Once your on-demand cluster is running smoothly, resist the temptation to increase it by purchasing a lot of desktop computers you don't otherwise need. The use of LTSP with desktop computers is cost effective only because you already paid for them. There is no financial outlay to acquire them, install them or maintain them when any of their components fail. Dedicated multiprocessor rackmount computers are easily the cheapest way to add processing power to a cluster. By omitting the unnecessary peripherals, they also save money, power, cooling and some failures.

OpenMosix or Mosix offer a quick and easy way to get cluster benefits, but the kernel is making migration decisions in real time. It is inherently less efficient than using explicit workload management with processes dedicated to individual nodes and communicating using MPI. Because you can support both Mosix and MPI within the same cluster, you may want to add job control and MPI libraries to the LTSP client filesystem. Applications that are cluster-aware take advantage of MPI and achieve the ultimate performance available. The other applications always gain partial benefits from Mosix.

On a dual-MPI/Mosix cluster, users have the incentive to migrate to MPI applications. The load balancing algorithms of Mosix always give priority to a local MPI process over a migrated Mosix process, so cluster-unaware applications run more slowly. We haven't started using MPI yet, because none of our critical engineering applications would benefit from it enough to justify the effort needed to establish it.

Further Logic Plans

Our next step in supporting QM's logic simulation needs is to use co-simulation, in which a regression test runs in real time on programmable logic chips. The testing speed is impressive too, because it eliminates the factor of a million speed ratio of simulation. Allowing for the co-simulation support logic, which also has to be placed in the programmable chip, about 10% of the logic can be tested at once. Therefore, each chip can execute tests as fast as a 50,000-node cluster.

No changes to the Linux and cluster configuration are necessary, but open-source tools are critical to keeping the process simple. Every test has to be processed by the place-and-route tools before execution, the test benches have to be written in a special way and a new level of data organization tracks all work in progress.

Conclusion

LTSP runs well within a Windows network and makes it easy to deploy software temporarily across the whole company without modifying the hard drives. Deploying Icarus on the OpenMosix cluster saved months of development time and ensured a more reliable product. The flexibility of open-source components increased our productivity, and the availability of our cluster enhances our corporate capabilities.

Resources for this article: /article/7553.

Dr Alexander Perry (alex.perry@qm.com) is principal engineer at Quantum Magnetics in San Diego, California.

Hoke Trammell (hoke.trammell@qm.com) is a staff scientist for Electromagnetic Sensing at Quantum Magnetics.

David Haynes (david.haynes@qm.com) is corporate Network Administrator at Quantum Magnetics.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

uClinux for Linux Programmers

David McCullough

Issue #123, July 2004

Adapt your software to run on processors without memory management—it's easier than you think.

uClinux has seen a huge increase in popularity and is appearing in more commodity devices than ever before. Its use in routers (Figure 1), Web cameras and even DVD players is testimony to its versatility. The explosion of low-cost, 32-bit CPUs capable of running uClinux is providing even more options to manufacturers considering uClinux. Now with uClinux's debut as part of the 2.6 kernel, it is set to become even more popular.

Figure 1. The SnapGear LITE2 VPN/Router runs uClinux.

With more embedded developers facing the possibility of working with uClinux, a guide to its differences from Linux and its traps and pitfalls is an invaluable tool. Here we discuss the changes a developer might encounter when using uClinux and how the environment steers the development process.

No Memory Management

The defining and most prevalent difference between uClinux and other Linux systems is the lack of memory management. Under Linux, memory management is achieved through the use of virtual memory (VM). uClinux was created for systems that do not support VM. As VM usually is implemented using a processing unit called an MMU (memory management unit), you often hear the term NOMMU when traveling in uClinux circles.

With VM, all processes run at the same address, albeit a virtual one, and the VM system takes care of what physical memory is mapped to these locations. So even though the virtual memory the process sees is contiguous, the physical memory it occupies can be scattered around. Some of it even may be on a hard disk in swap. Because arbitrarily located memory can be mapped to anywhere in the process' address space, it is possible to add memory to an already running process.

Without VM, each process must be located at a place in memory where it can be run. In the simplest case, this area of memory must be contiguous. Generally, it cannot be expanded as there may be other processes above and below it. This means that a process in uClinux cannot increase the size of its available memory at runtime as a traditional Linux process would.

Although all programs need to be relocated at run time so that they can execute, it is a fairly transparent task for the developer. It is the direct effect of no VM that is the thorn in every uClinux developer's side. The net effect is that no memory protection of any kind is offered—it is possible for any application or the kernel to corrupt any part of the system. Some CPU architectures allow certain I/O areas, instructions and memory regions to be protected from user programs but that is not guaranteed. Even worse than the corruption that crashes a system is the corruption that goes unnoticed, and tracking down random interprocess corruption can be extremely difficult.

Without VM, swap is effectively impossible, although this limitation is rarely an issue on the kinds of systems that run uClinux. They often do not have hard drives or enough memory to make swap worthwhile.

Kernel Differences

To a kernel developer, uClinux offers little in the way of differences from Linux. The only real issue is that you cannot take advantage of the paging support provided by an MMU. In practice, this doesn't affect much of the kernel. tmpfs, for example, does not work on uClinux because it relies on the VM system.

Similarly, all of the standard executable formats are unsupported, because they make use of VM features that do not exist under uClinux. Instead, a new format is required, the flat format. Flat format is a condensed executable format that stores only executable code and data, along with the relocations needed to load the executable into any location in memory.

Device drivers often need some work when you move to uClinux, not because of differences in the kernels, but due to the kinds of devices the kernel needs to support. For example, the SMC network driver supports ISA SMC cards. They usually are 16-bit and are located at I/O addresses below 0x3ff. The same driver

easily can be made to support the non-ISA embedded versions of the chip, but it may need to run in 8-, 16- or 32-bit mode, at an I/O address that is a full 32-bit address and at an interrupt number quite often higher than ISA's maximum of 16. So despite the fact that the bulk of the driver is the same, the hardware specifics can require a little porting effort. Quite often, older drivers store I/O addresses in short format, which does not work on an embedded uClinux platform with devices appearing at memory-mapped I/O addresses.

The implementation of mmap within the kernel is also quite different. Though often transparent to the developer, it needs to be understood so it is not used in ways that are particularly inefficient on uClinux systems. Unless the uClinux mmap can point directly to the file within the filesystem, thereby guaranteeing that it is sequential and contiguous, it must allocate memory and copy the data into the allocated memory. The ingredients for efficient mmap usage under uClinux are quite specific. First, the only filesystem that currently guarantees that files are stored contiguously is romfs. So one must use romfs to avoid the allocation. Second, only read-only mappings can be shared, which means a mapping must be read-only in order to avoid the allocation of memory. The developer under uClinux cannot take advantage of copy-on-write features for this reason. The kernel also must consider the filesystem to be “in ROM”, which means a nominally read-only area within the CPU's address space. This is possible if the filesystem is present somewhere in RAM or ROM, both of which are addressable directly by the CPU. One cannot have a zero allocation mmap if the filesystem is on a hard disk, even if it is a romfs filesystem, as the contents are not directly addressable by the CPU.

Memory Allocation (Kernel and Application)

uClinux offers a choice of two kernel memory allocators. At first it may not seem obvious why an alternative kernel memory allocator is needed, but in small uClinux systems the difference is painfully apparent. The default kernel allocator under Linux uses a power-of-two allocation method. This helps it operate faster and quickly find memory areas of the correct size to satisfy allocation requests. Unfortunately, under uClinux, applications must be loaded into memory that is set aside by this allocator. To understand the ramifications of this, especially for large allocations, consider that an application requiring a 33KB allocation in order to be loaded actually allocates to the next power of two, which is 64KB. The 31KB of extra space allocated cannot be utilized effectively. This order of memory wastage is unacceptable on most uClinux systems. To combat this problem, an alternative memory allocator has been created for the uClinux kernels. It commonly is known as either page_alloc2 or kmalloc2, depending on the kernel version.

page_alloc2 addresses the power-of-two allocation wastage by using a power-of-two allocator for allocations up to one page in size (a page is 4,096 bytes, or

4KB). It then allocates memory rounded up to the nearest page. For the previous example, an application of 33KB actually has 36KB allocated to it; a savings of 28KB for a 33KB application is possible.

page_alloc2 also takes steps to avoid fragmenting memory. It allocates all amounts of two pages (8KB) or less from the start of memory up and all larger amounts from the end of free memory down. This stops transient allocations for network buffers and so on, fragmenting memory and preventing large applications from running. For a more detailed example of memory fragmentation, see the example in the Applications and Processes section below. page_alloc2 is not perfect, but it works well in practice, as the embedded environments that run uClinux tend to have a relatively static group of long-lived applications.

Once the developer gets past the kernel memory allocation differences, the real changes appear in the application space. This is where the full impact of uClinux's lack of VM is realized. The first major difference most likely to cause an application to fail under uClinux is the lack of a dynamic stack. On VM Linux, whenever an application tries to write off the top of the stack, an exception is flagged and some more memory is mapped in at the top of the stack to allow the stack to grow. Under uClinux, no such luxury is available as the stack must be allocated at compile time. This means that the developer, who previously was oblivious to stack usage within the application, must now be aware of the stack requirements. The first thing a developer should consider when faced with strange crashes or behavior of a newly ported application is the allocated stack size. By default, the uClinux toolchains allocate 4KB for the stack, which is close to nothing for modern applications. The developer should try increasing the stack size with one of the following methods:

1. Add `FLTFLAGS = -s <stacksize>` and `export FLTFLAGS` to the Makefile for the application before building.
2. Run `flthdr -s <stacksize> executable` after the application has been built.

The second major difference that strikes a uClinux developer is the lack of a dynamic heap, the area used to satisfy memory allocations with malloc and related functions in C. On Linux with VM, an application can increase its process size, allowing it to have a dynamic heap. This traditionally is implemented at the low level using the sbrk/brk system calls, which increase/change the size of a process' address space. The heap's management by library functions such as malloc then is performed on the extra memory obtained by calling sbrk() on behalf of the application. If an application needs more memory at any point, it can get more simply by calling sbrk() again; it also can decrease memory using brk(). sbrk() works by adding more memory to the end of a process (increasing

its size). brk() arbitrarily can set the end of the process to be closer to the start of the process (reduce the process size) or further away (increase the process size).

Because uClinux cannot implement the functionality of brk and sbrk, it instead implements a global memory pool that basically is the kernel's free memory pool. There are pitfalls with this method. For example, a runaway process can use all of the system's available memory. Allocating from the system pool is not compatible with sbrk and brk, as they require memory to be added to the end of a process' address space. Thus, a normal malloc implementation is no good, and a new implementation is needed.

A global pool approach has some advantages. First, only the amount of memory actually required is used, unlike the pre-allocated heap system that some embedded systems use. This is extremely important on uClinux systems, which generally are running with little memory. Another advantage is that memory can be returned to the global pool as soon as it is finished being used, and the implementation can take advantage of the existing in-kernel allocator for managing this memory, reducing the size of application code.

One of the common problems new users encounter is the missing memory problem. The system is showing a large amount of free memory, but an application cannot allocate a buffer of size X. The problem here is memory fragmentation, and all of the uClinux solutions available at this time suffer from it. Because of the lack of VM in the uClinux environment, it is nearly impossible to utilize memory fully due to fragmentation. This is best explained by example. Suppose a system has 500KB of free memory and one wishes to allocate 100KB to load an application. It is easy to think that this would be possible. However, it is important to remember that one must have a contiguous 100KB block of memory in order to satisfy the allocation. Suppose the memory map looks like this. Each character represents approximately 20KB, and X marks areas allocated or in use by other programs or by the kernel:

```
    0    100   200   300  400   500   600   700  800   900   1000
   -+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+--
    |XXXXX|XXXXX|---XX|--X--|-X---|XX---|-X---|-XX--|-X---|XXXXX|
```

In this case, 500KB are free, but the largest contiguous block is only 80KB. There are many ways to arrive at such a situation. A program that allocates some memory and then frees most of it, leaving a small allocation in the middle of a larger free block, often is the cause. Transient programs under uClinux also can affect where and how memory is allocated. The uClinux page_alloc2 kernel allocator has a configuration option that can help identify this problem. It enables a new /proc entry, /proc/mem_map, that shows pages and their

allocation grouping. Documenting this is beyond the scope of this article, but more information can be found in the kernel source for page_alloc2.c.

The question is often asked, why can't this memory be defragmented so it is possible to load a 100KB application? The problem is that we don't have VM and we cannot move memory being used by programs. Programs usually have references to addresses within the allocated memory regions, and without VM to make the memory always appear to be at the correct address, the program will crash if we move its memory. There is no solution to this problem under uClinux. The developer needs to be aware of the problem and, where possible, try to utilize smaller allocation blocks.

Applications and Processes

Another difference between VM Linux and uClinux is the lack of the fork() system call. This can require quite a lot of work on the developer's part when porting applications that use fork(). The only option under uClinux is to use vfork(). Although vfork() shares many properties with fork(), the differences are what matter the most.

fork() and vfork(), for those unfamiliar with these system calls, allow a process to split into two processes, a parent and a child. A process can split many times to create multiple children. When a process calls fork(), the child is a duplicate of the parent in all ways, but it shares nothing with the parent and can operate independently, as can the parent. With vfork() this is not the case. First, the parent is suspended and cannot continue executing until the child exits or calls exec(), the system call used to start a new application. The child, directly after returning from vfork(), is running on the parent's stack and is using the parent's memory and data. This means the child can corrupt the data structures or the stack in the parent, resulting in failure. This is avoided by ensuring that the child never returns from the current stack frame once vfork() has been called and that it calls _exit when finishing—exit cannot be called as it changes data structures in the parent. The child also must avoid changing any information in global data structures or variables, as such changes may break the execution of the parent.

Making an application use vfork instead of fork usually falls into the absolutely simple or incredibly difficult category. Generally, if the application does not fork and then exec() almost immediately, it needs to be checked carefully before fork() can be replaced with vfork().

The uClinux flat executable format, though it doesn't directly affect applications and their operations, does allow quite a few options that the usual ELF executables under Linux do not. Flat format executables come in two basic flavors, fully relocated and a variation of position-independent code (PIC). The

fully relocated version has relocations for its code and data, while the PIC version generally needs only a few relocations for its data.

One of the most advantageous features to the embedded developer is execute-in-place (XIP). This is where the application executes directly from Flash or ROM, requiring the absolute minimum of memory, because only the memory for the data of the application is needed. This allows the text or code portion to be shared between multiple instances of the application. Not all uClinux platforms are capable of XIP, as it requires compiler support and the PIC form of the flat executable. So unless the toolchain for a given platform can do PIC, it cannot do XIP. Currently, only the m68k and ARM toolchains provide the required level of support for flat format XIP. romfs is the only filesystem to support XIP under uClinux, because the application must be stored contiguously within the filesystem for XIP to be possible.

The flat format also defines the stack size for an application as a field in the flat header. To increase the stack allocated to an application, a simple change of this field is all that is required. This can be done with the flthdr command, like this:

```
flthdr -s  flat-executable
```

The flat format also allows two compression options. The entire executable can be compressed, providing maximum ROM savings. It also offers the often useful side effect that the application is loaded entirely into a contiguous RAM block. You also may choose data-segment-only compression. This is important if you want to save ROM space but still want the option to utilize XIP. The following:

```
flthdr -z flat-executable
```

creates a fully compressed executable, and

```
flthdr -d flat-executable
```

compresses only the data segment.

Shared Libraries

Although a complete discussion of shared libraries is beyond the scope of this article, they are quite different under uClinux. The currently available solutions require compiler changes and care on the part of the developer. The best way to create shared libraries is to start with an example. The current uClinux distributions provide shared libraries for both the uC-libc and uClibc libraries.

The method for creating a shared library isn't difficult, and both of these libraries provide a good, clean example of how it is done. To set expectations appropriately, the GCC -shared option is not part of the shared library creation process, so do not expect it to be familiar. Shared libraries under uClinux are flat format executables, just like applications, and to be truly shared must be compiled for XIP. Without XIP, shared libraries result in a full copy of the library for each application using it, which is worse than statically linking your applications.

Summary

The step into uClinux from Linux often is more than the differences between uClinux and Linux. uClinux systems tend to be more deeply embedded systems, with smaller memories and ROM footprints and an unusual array of devices. The loss of a hard drive and the tight resource limits, coupled with no memory protection and a number of other subtle differences can make a developer's first adventure into uClinux more difficult than imagined. The best way to get started is to look at the uClinux Emulators (Figure 2) and cheap hardware (Figure 3) options available.

Figure 2. uClinux Running under Xcopilot (Palm Emulator)

Figure 3. uClinux Running on a Real Palm IIIx (with Microwindows)

Hopefully, highlighting these issues will help the wary developer be prepared beforehand and avoid some of the common pitfalls and misconceptions of working with uClinux.

Resources for this article: /article/7546.

David McCullough is a senior software engineer and a veteran embedded software developer. Prior to working at SnapGear and Lineo, he held software development and engineering management positions at Stallion Technologies and was involved in the development of products based on SCO and BSD UNIX. David ported and maintained XFree86 on SCO UNIX for several years and recently was instrumental in the development of the uClinux port of Linux 2.6.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

At the Forge

Slash

Reuven M. Lerner

Issue #123, July 2004

One of the oldest and most full-featured Web community systems is also among the hardest to install. Reuven gets you started as a Slash Webmaster with advice on mod_perl and other prerequisites.

The popularity of Weblogs, also known as blogs, has been growing for several years and shows no signs of letting up. Although many Webloggers continue to record their thoughts using third-party services, such as LiveJournal and Blogger, running a Weblog on your own server is becoming easier to do. Over the last few months, we have looked at several different packages that provide this functionality, including COREBlog, a Zope product, and Blosxom, a set of CGI programs written in Perl.

Last month, we looked at a slightly different type of system for Weblogs when we examined XOOPS. XOOPS, like its cousins PHPNuke and PostNuke, allows users to create content management and on-line communities as well as administer users and groups. XOOPS makes it easy to give each user on a system his or her own personal Weblog.

As popular as XOOPS may be, the undisputed granddaddy of community software is Slash, which powers the popular Slashdot.org and use.perl.org Web sites. Slash primarily is used to disseminate news articles and comments, but it has a powerful Weblog feature that is available to every user on the system. Some would argue that Slashdot itself is a community-authored Weblog, an argument I also find somewhat convincing. Best of all, this Weblog feature is combined into other elements of the site. Thus, when you see a particularly insightful or insipid comment from another user, you immediately can view that user's journal to learn more.

This month, we look at the installation and configuration of a simple Slash site, with built-in support for users and Weblogs, or journals, as they are called in Slash lingo. Next month, we will take a closer look at the Weblog functionality available on Slash, as well as how to configure and change it to suit our particular needs.

Infrastructure

The main distribution and discussion site for Slash is Slashcode. As I write this in early April 2004, the most recent posting is titled, "Making Slash Install Friendly", in which the author asks if there are clear and simple directions for installing Slash. Unfortunately, the answer appears to be no. This is the case for several reasons:

- The Slash documentation and instructions point most people to the officially released .tar.gz packages, and these packages are more than two years out of date.
- The latest CVS version is freely available and up to date but potentially unstable, unless you know which tagged version to grab.
- The installation is a fairly manual process, with room for error at several critical points.
- The documentation for Slash is somewhat lacking in quality.
- Even if you get a recent, working copy of the code installed, full Slash installation requires installing mod_perl, Template Toolkit, MySQL and a number of other Perl modules and standalone tools, each having its own slightly odd and nonstandard options.

If you are an expert with mod_perl, MySQL and Apache, installing Slash is a slightly annoying but doable process. If you are less than an expert, the results probably are worthwhile, but you should expect to learn quite a bit about each of these technologies along the way. You may find yourself turning to the IRC channel and Web site for support and ideas.

The first step in installing Slash is to install Apache, mod_perl and MySQL. Any modern version of MySQL works fine; the real problems are with Apache and mod_perl, which can be tricky for first-timers to install. Luckily, this part of the installation process has not changed significantly over the years, meaning that you can follow the MySQL, Apache and mod_perl installation instructions at the InstallSlash site (see the on-line Resources section). If neither Perl nor expat are installed on your system, you should follow the InstallSlash instructions for those as well. Remember that on Red Hat and Fedora systems, you need to install not only the expat RPM but the expat-devel RPM too. You also need to define a MySQL database into which site information can be stored; by default, this is called slashdb.

A major source of trouble when compiling Apache and mod_perl is the need to define EVERYTHING=1. This activates all of mod_perl's hooks, allowing mod_perl to override all of Apache's default behavior, including authentication, authorization, URL rewriting and logging. Without defining EVERYTHING=1, mod_perl can generate only content. If your system came with mod_perl installed, it probably was not compiled with EVERYTHING=1 defined, meaning that you need to compile it again by yourself.

The Slash instructions also advise system administrators to set PERL_MARK_WHERE=1 when compiling, although the mod_perl code and documentation indicate this directive removes most undefined value warnings from the error log. I ignored this suggestion and used Slash on my existing mod_perl installation, and I did not notice any ill effects.

Finally, you should install the Bundle::Slash package from CPAN (a worldwide network of servers containing freely available Perl modules and documentation) on your system, using the automatic CPAN tools. Actually, Bundle::Slash does not contain any code; it lists the modules you need in order to run Slash. In this way, it saves you from having to remember (and type in) all of the Slash-related modules that must be installed. You can install these modules while logged in as root by typing:

```
# perl -MCPAN -e 'install Bundle::Slash'
```

If you never have used CPAN before, you will be asked to define a number of CPAN-related parameters, including the closest CPAN archive from which you can retrieve modules. This can be a bit tricky the first time, although you probably can accept the defaults without suffering any consequences.

One complicated part about installing these CPAN modules is DBIx::Password, which asks for some site-specific information when it is installed. The InstallSlash directions indicate what you should type in response to the prompts. If you decide to change values for reasons of security or personal taste, be sure to remember what names you used.

The most difficult part of the Bundle::Slash download is the Template Toolkit installation. Template Toolkit is a popular and powerful templating system for mod_perl, and it is used to display the various pages within a Slash site in a consistent and efficient manner.

Retrieving a Tagged Version from CVS

At this point, you need to download and install the Slash code itself. The InstallSlash instructions walk you through the download and installation process for the latest packaged version (2.2.6), which is on SourceForge.

However, as I indicated above, a great deal of development has happened in the two years since 2.2.6 was released, and those improvements are available only in the CVS version. You can retrieve the latest version from CVS and save it in a directory named slash with the following command:

```
cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:
↪/cvsroot/slashcode co -r T_2_3_0_148 slash
```

At the same time, checking out the latest CVS version from SourceForge has its own pitfalls, because you run the risk of installing code recently checked in but not tested. The best solution is to use a tagged version. As experienced CVS developers know, each file in CVS has its own revision number, such as 1.5 or 2.8 or 3.1.1.2. These revision numbers reflect the revision of the individual file and have nothing to do with the overall project. Thus, although a software project might be released to the public as version 2.0, the individual files almost certainly do not have revision numbers of 2.0. To assign a collective name (or number) to the current state of all files in a project, you must use tags, sometimes known as symbolic revisions. Assuming that you have write privileges, you can tag all of the files in the current directory release-2-0 with the following:

```
cvs tag release-2-0
```

Tags cannot contain certain characters, including periods and commas. This is why we needed to use a hyphen rather than a period to indicate release 2.0 of our software. This raises the question of which tag you can or should retrieve. In the case of Slash, the code is moving slowly toward the 2.3.0 release, and the developers have standardized on tags that look like T_2_3_0_XXX, where T stands for tested and XXX is incremented with each new tag. As of this writing, the latest tag is T_2_3_0_148; it can be retrieved with:

```
cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:
↪/cvsroot/slashcode co -r T_2_3_0_148 slash
```

This command creates a directory named slash and puts all of the Slash-related code, libraries and documentation into it. Particularly helpful is the INSTALL file, which is an up-to-date version of the directions designed to work with the CVS version you just checked out. I followed the directions in this file, typing `make install` to install all of the Slash components. A message from the installation procedure indicates what command to add to the end of the Apache configuration file. This ensures that Apache includes the appropriate Perl modules at startup, reducing mod_perl's memory footprint and execution time significantly.

One of the nice things about Slash is that it can handle multiple sites easily using the same code. That is, if you decide you want to have separate on-line

communities for Perl, Python, Tcl and Ruby, Slash can handle this for you. Each community needs its own hostname, but they can be totally distinct from one another. In other words, installing a Slash site is a separate procedure from installing the Slash software. Once the software is installed, in /usr/local/slash by default, you can create a new site by running:

```
/usr/local/slash/bin/install-slashsite -u USER
```

Here, USER is the same virtual user you created back when you installed DBIx::Password. By default, this user is virtslash. install-slashsite asks several more questions, including the administrator user name and password, and then tells you to stop and start Apache with the apachectl program, typically installed in /usr/local/apache/sbin. You should *not* use apachectl's restart command. Especially when working with mod_perl, you should always stop Apache, pause a few seconds to allow the processes to exit completely, and then start it again.

When I tried to run `install-slashsite`, I discovered that at least one CPAN module (LWP::Parallel::UserAgent) was missing. It did not take too long to install it, but I was frustrated to discover that neither Bundle::Slash nor make install noticed or fixed this.

A Simple Slash Site

Your Slash site now is ready for use. On my machine, named chaim-weizmann on my private home network, I was able to view my site by going to http://chaim-weizmann.

Of course, a newly installed Slash site is not very exciting. If you log in using the administrator user name and password you gave to install-slashsite, you are given a number of options and menus that provide a fair amount of control over the system. Many of the menus are hard to find, and it takes some time before a new Slash administrator will understand where different changes are made, either on the Web site or in the on-disk templates and programs. A good introduction to Slash-site maintenance, slashguide.pod, is in the docs directory that comes with the CVS version of Slash.

But, if you are at all familiar with the original Slashdot site, you should be able to start posting stories and soliciting comments right away. While logged in as the site administrator, click on the New link at the top of the home page. Enter a topic, title, department, intro copy (displayed on the home page) and extended copy (displayed on the individual story page). You even can set up a poll or link a story to an existing poll. Once the story is approved, it is visible to the entire world. Visitors to the site optionally can be allowed to comment and

even can act as moderators. Indeed, the community moderation and meta-moderation of Slash stories is one of the most intriguing ideas that Slash has brought to the table.

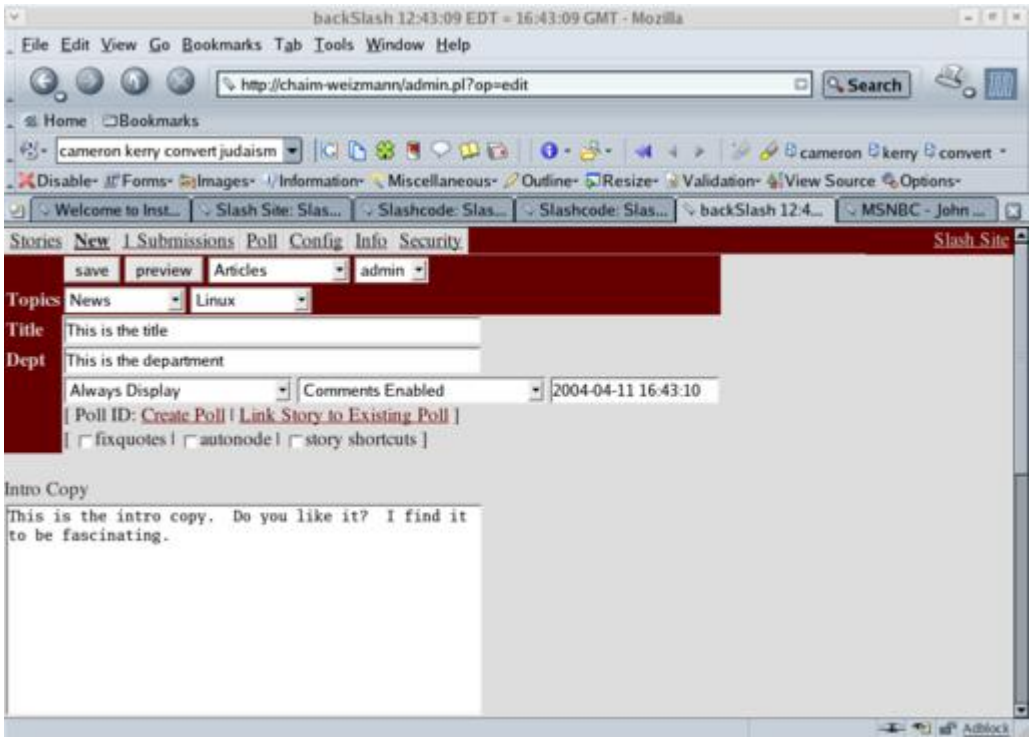


Figure 1. Enter the text here...

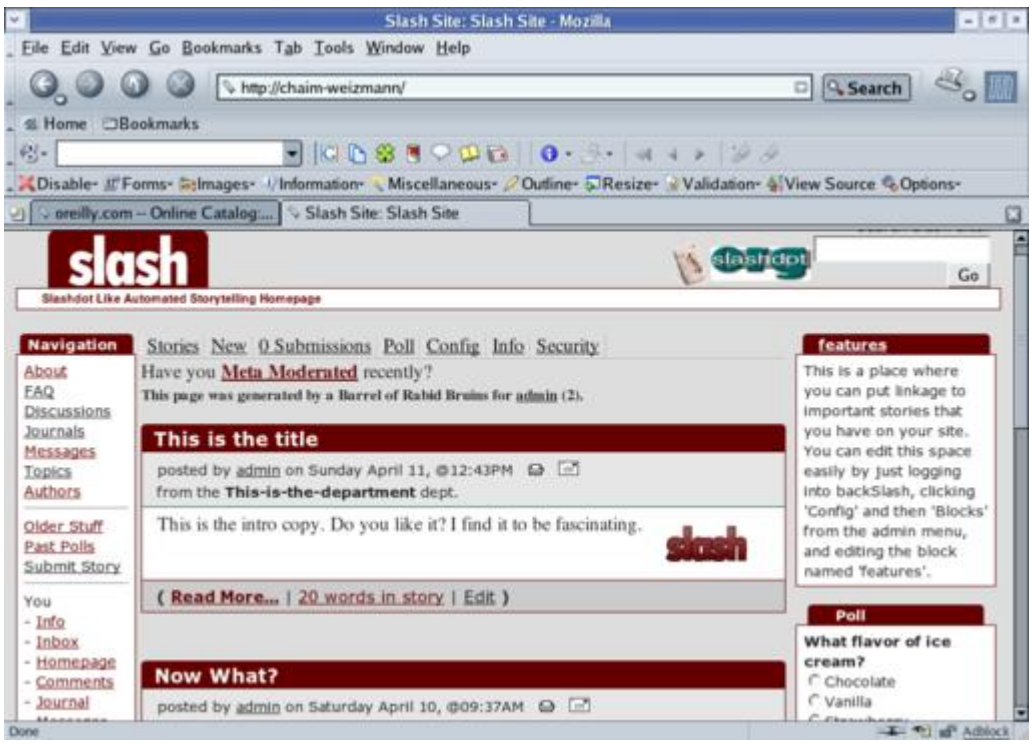


Figure 2. ...and it appears like this.

Conclusion

Slash is more difficult to install than any of the other packages we have discussed to date. This is partly due to the authors' choice of technology, because mod_perl inherently is more difficult to install than is PHP, used by XOOPS, or Zope. At the same time, Slash provides more community and Weblog functions than the other packages, is known to handle a high load and continues to be maintained at a fairly high level.

If you aren't afraid to get your hands dirty, and if you want the functionality the Slashdot site provides, Slash may be a good choice for Weblogs written by a community or by individuals. Next month, we will look at the personal journals that Slash allows people to make, along with the friend system that allows people to categorize other users on the system, creating a virtual community that reflects the real world.

Resources for this article: /article/7545.

Reuven M. Lerner, a longtime Web/database consultant and developer, is now a first-year graduate student in the Learning Sciences program at Northwestern University. His Weblog is at altneuland.lerner.co.il, and you can reach him at reuven@lerner.co.il.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Cooking with Linux

It's a Cross Platform, All Right!

Marcel Gagné

Issue #123, July 2004

Linux-based file managers make it easy to use files and printers on your legacy Microsoft Windows systems.

Yes, I admit it, François. That's very funny. When I told you that this month's theme was cross-platform development, I didn't mean platforms that make you cross, although I can understand thinking of some OSes as cross platforms. As amusing as those images are, I think the artwork you chose for the menus tonight might raise a few eyebrows, despite our, shall we say, sympathetic audience.

Speaking of which, I see that our guests have arrived. Welcome, *mes amis*, to *Chez Marcel*, home of exceptional Linux fare, one of the world's greatest wine cellars and the greatest customers in the world. Make yourselves comfortable. François and I were discussing the theme of this month's issue, cross-platform development, and my waiter was getting a little rambunctious. It almost seems as though we should bring out a white Zinfandel for this, but luckily, we have none in stock. François, to the cellar, *immédiatement*! Bring up the 1992 Napa Valley Cabernet Sauvignon. *Vite!*

As many of you know, Microsoft Windows remains a part of the average business IT department. Many of us must be able to exchange information between Windows and Linux. For example, somehow you convinced management to let you run Linux on your workstations instead of Windows. Maybe you are using your own notebook. Whatever the reason, you now are going to have to deal with the Windows workgroup or domain and the appropriate shared files and printers. Although Jon in accounting isn't particularly fond of his Windows XP box, many important files are shared from that machine, files that are shared in the network neighborhood.

You might ask yourself how easy it is to take advantage of the network neighborhood. This is an interesting question considering how many file and print servers out there aren't running Windows but Linux, providing the file sharing experience through Samba. Consequently, it isn't surprising that Samba-compatible client software tends to come as part of the standard installation on most modern Linux distributions. What that means is you can connect to a Windows share on the network using the smbclient program, started with `smbclient -L sedna`, which would produce a report of shares that looks something like this:

```
Domain=[ACCOUNTING] OS=[Windows 5.1] Server=[Windows 2000 LAN Manager]

        Sharename      Type      Comment
        ---------      ----      -------
        SEDNA_C        Disk
        IPC$           IPC       Remote IPC
        Reports        Disk
        Policies       Disk
```

Assuming you had permission to view the Reports folder, you could connect to it like this:

```
smbclient //sedna/reports -U winuser
```

In the above example, I am connecting to a Windows XP box from my Linux workstation as user winuser. The system then asks me for a password, after which I am taken to a Samba prompt that looks like this:

```
Domain=[ACCOUNTING] OS=[Windows 5.1] Server=[Windows 2000 LAN Manager]
smb: \>
```

From there, type `help` and smbclient offers up a list of all the things you can do while connected using the commands at your disposal. A number of them are obvious things, such as `dir`, `copy` and so on. Although all of this is good, it's not pretty in the graphical sense and you can't use this while working in your graphical file managers or inside OpenOffice.org's applications.

Whether you are using KDE or GNOME as your desktop environment, rest assured that all the tools you need to join the network neighborhood are right there at your disposal. Best of all, it's extremely easy to do. Let's start by taking a look at Konqueror.

Open up Konqueror (either as the file manager or the browser) and type `smb:/` in the Location field.

Samba servers or Windows machines advertising network shares first appear in the browser window under their workgroup name (for example, ACCOUNTING, SALESGRP). Figure 1 shows a Konqueror session with a two-panel view; click

Window on the menubar and select Split View, Left/Right. In the left panel, we have the basic network browser view with three active workgroups. In the right-hand panel, I've clicked on the ACCOUNTING workgroup to show the computers belonging to that group.

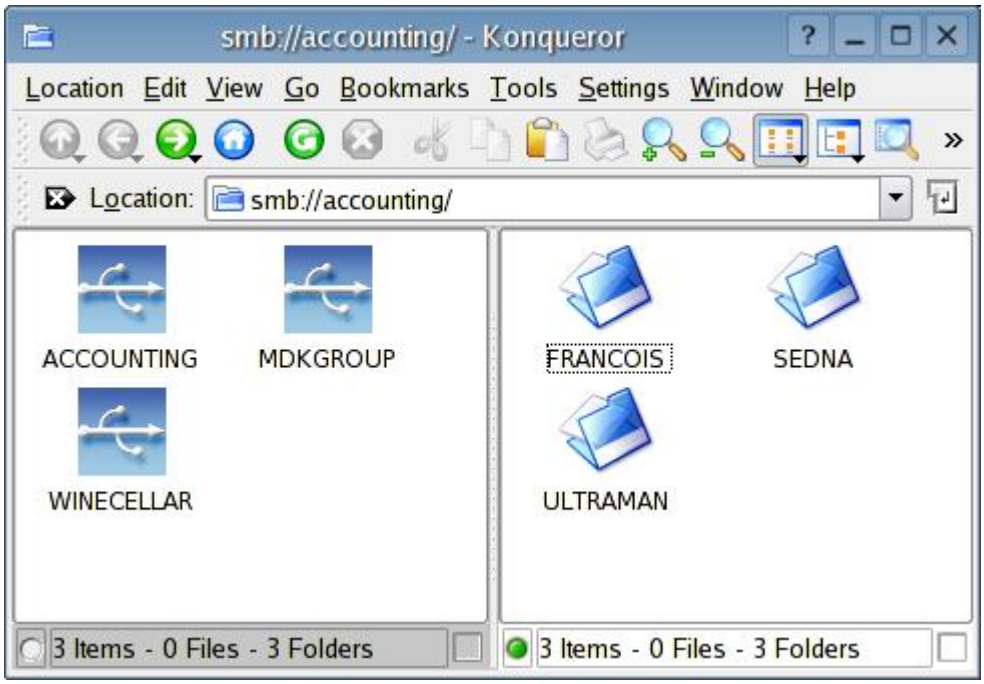


Figure 1. Using Konqueror to Browse Workgroups

To read, write or otherwise make use of the files shared on those computers, double-click on the corresponding folder for that computer—François' computer for instance. All of the available shared directories (or folders) then are visible (Figure 2).

Figure 2. Once connected, browsing is basic file manager navigation.

From here, it's all classic drag and drop, graphical file manager browsing. By clicking (or double-clicking, depending on the configuration), I can go inside the cooking folder, locate the appropriate document and open it with OpenOffice.org Writer if I choose. (Right-click on the file, select Open with and *voilà*, I'm editing a document on a shared Windows resource.)

Ideally, we don't want to go through this whole navigation process each and every time. To bring a given network share a few clicks closer, simply bookmark the appropriate shared folder.

Over on the GNOME side of things, we have Nautilus. You should find the process similar to what we went through with Konqueror. Start Nautilus and type `smb:///` in the Location bar. Nautilus then displays the active workgroups on the network (Figure 3).

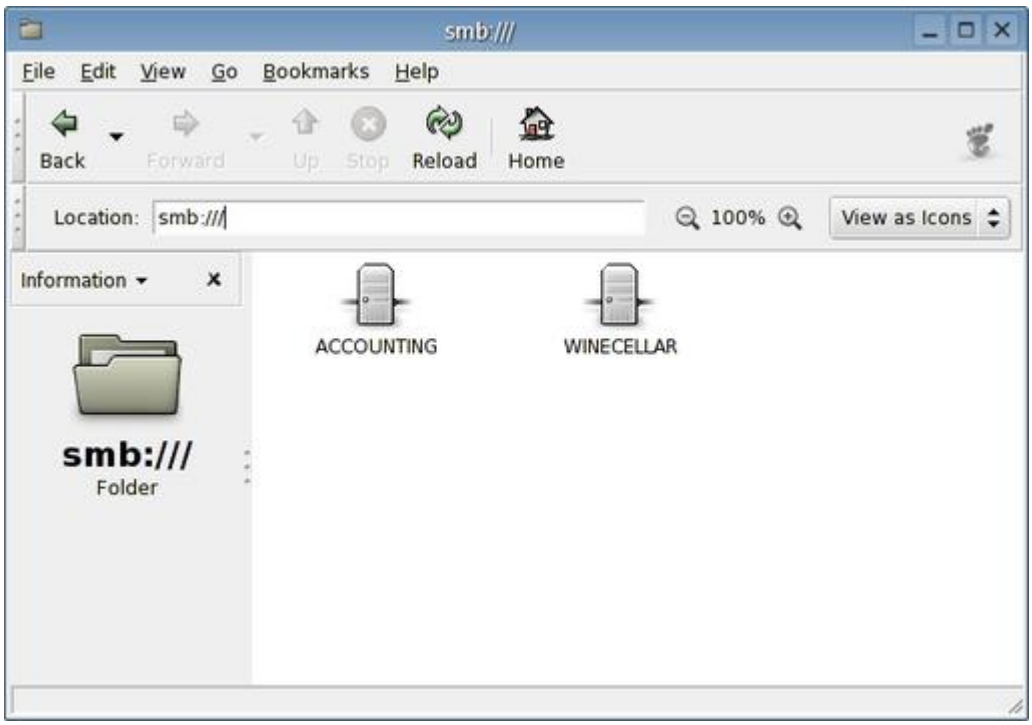


Figure 3. Nautilus in SMB Network Browse Mode

From there, you can double-click on one of the workgroups to select a computer. Then, from the list of computers, double-click on your choice, and you can browse the individual resources offered (Figure 4). Be aware that when you move around from computer to computer like this, you occasionally may be asked for a user name and password for that computer or even the specific folder.

Figure 4. Windows and Samba shares are accessed easily.

As with the Konqueror example before this, you can save yourself a little time by bookmarking the folder of your choice. The problem with both of the suggestions I've made is that neither of them lets you permanently mount network drives. Accessing a particular folder requires that you do a little command-line work, an easy enough process but not quite the point-and-click ease that Windows users on your network want to see. Let's have François refill our glasses while we take a look at a way to solve this dilemma.

For a more robust and flexible means of working in and with the network neighborhood, you simply must take a look at Smb4K, a super-classy SMB browser tool that also is flexible and powerful. Furthermore, Smb4K makes it possible to preview shares, mount shares locally without needing to run as root, reconnect automatically on startup and more.

At the time of this writing, Smb4K was sitting at the 0.3.2 release, but I found it to be a capable package and definitely worth the time it takes to investigate. Binary packages for Debian, SuSE and Fedora are available from the site, as is the full source. Building Smb4K from source is as easy as the classic extract and build five-step:

```
tar -xzvf smb4k-0.3.2.tar.gz
cd smb4k-0.3.2
./configure --prefix=/usr
make
su -c "make install"
```

When you have installed the package, run the program by calling `smb4k`. As soon as you start Smb4K, it scans the network looking for active shares. You can fine-tune its functionality, including such options as whether you want shares to be reconnected automatically by clicking on Settings in the menubar and selecting Configure Smb4K. The graphical interface is intuitive and easy to navigate, and the package as a whole is easy to use.

The display is divided into a left-hand side navigation panel where workgroups, computers and shares can be listed and navigated. To mount a share, right-click on it and select mount. If you would rather see what you are getting into first, choose Preview instead.

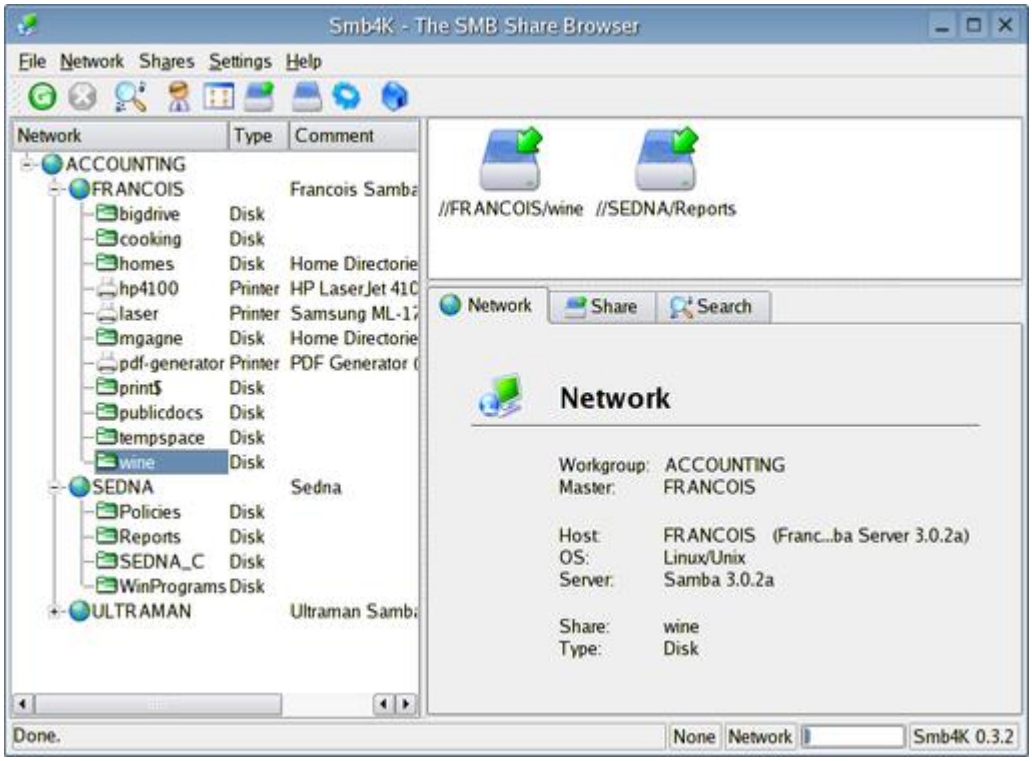


Figure 5. Smb4K: could this be the ultimate SMB browser?

Mounted drives appear in the top right-hand window as drive icons. Double-clicking on one of the drive icons calls Konqueror. If you run `df` from the command line, you see that the drives now are mounted for your use in your own home directory under an Smb4K directory prefix. For instance, for the example in Figure 5, the listing looks like this:

```
Filesystem       Size Used Avail Use% Mounted on
//SEDNA/Reports  4.0G 3.0G  1.1G  75% /home/marcel/smb4k/SEDNA/Reports
//FRANCOIS/wine   13G 8.8G  3.3G  73% /home/marcel/smb4k/FRANCOIS/wine
```

Now, any of my applications—whether KDE, GNOME, shell-based or anything else—can access the shares. Being part of the neighborhood has never been easier.

Mon Dieu, mes amis, closing time has come so quickly. François, would you be so kind as to refill our guests' glasses a final time? We certainly don't want anyone going home cross, and with the items on today's menu, the neighborhood's doors all are open to you. Until next time, *mes amis*, let us all drink to one another's health. *A vï¿½re santï¿½ Bon appï¿½it!*

Resources for this article: /article/7548.

Marcel Gagné (mggagne@salmar.com) lives in Mississauga, Ontario. He is the author of *Moving to Linux: Kiss the Blue Screen of Death Goodbye!* (ISBN 0-321-15998-5) from Addison Wesley. His first book is the highly acclaimed *Linux System Administration: A User's Guide* (ISBN 0-201-71934-7). In real life, he is president of Salmar Consulting, Inc., a systems integration and network consulting firm.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Paranoid Penguin

Secure Anonymous FTP with vsftpd

Mick Bauer

Issue #123, July 2004

To keep your FTP site secure, stick to anonymous access only and run an FTP dæmon with minimal complexity.

Can you believe that in nearly four years of Paranoid Penguin columns, I've never talked about how to configure FTP services? This month I fix that, using my new favorite FTP server, Chris Evans' excellent vsftpd (Very Secure FTP Dæmon). Because my space here is limited and the best use of FTP is anonymous FTP, we focus on anonymous FTP. The FTP protocol's use of clear-text authentication makes it a terrible choice for anything but anonymous file transfer. But anonymous FTP is still plenty useful.

vsftpd is increasingly popular and is included with recent versions of Debian, SuSE, Fedora, Red Hat and other Linux distributions. This inclusion probably is because vsftpd provides a unique combination of security and convenience. It is easy to get up and running in a hurry, without having to make ugly security-vs.-expedience trade-offs.

Chris Evans created vsftpd with security as a central design goal, and its track record so far is impressive. In the nearly four years it's been available, as of this writing, vsftpd has had zero significant security vulnerabilities. Regardless of whether that's still true by the time you read this article, it speaks to vsftpd's excellent design philosophy, which borrows from OpenBSD's “secure by default, extra features disabled by default, minimal complexity overall” motto.

How minimalist is vsftpd? Its entire source tree is just over 1MB in size, fully uncompressed. The vsftpd executable itself is 80K.

Getting and Installing vsftpd

As I mentioned, vsftpd now is a standard package on many Linux distributions. The usual advantages of binary packages apply: convenience, easy patching and minimal impact on other system software. In Debian, SuSE, Fedora and Red Hat, the package you need predictably is named vsftpd. It has no particularly exotic dependencies. Most users probably will be perfectly happy with their distribution's stock vsftpd package.

If your distribution of choice doesn't provide a binary package for vsftpd, or if you need a later version than the one your distribution provides, compile vsftpd from its source code tarball, which is available at vsftpd.beasts.org. The build process is decidedly old school. If you aren't already, become root. Then, unpack the tarball and change your working directory to its root, like this:

```
# tar -xf vsftpd-1.2.1.tar.gz; cd vsftpd-1.2.1
```

Next, enter the command `make` without arguments. If it succeeds, there should be a vsftp executable in the current directory. Make sure the user nobody exists; if it doesn't, create it. vsftpd runs on this account.

Create the directory /usr/share/empty if it doesn't exist already. It should be owned by root and be neither group- nor world-writable—it will be used as the default vsftpd chroot jail.

Create a home directory for the anonymous FTP user. SuSE conventionally uses /srv/ftp, and other distributions use /var/ftp, but it can be whatever you like. Again, this directory should be owned by root and should not be writable by anyone else.

Create an anonymous FTP user account, such as ftp, and make sure its home directory is set to the one you created in the previous step. Your system already may have such an account. The anonymous ftp user should not be able to write in its home directory, and it should never own any files or directories.

Now you're ready to copy vsftpd and the vsftpd(8) and vsftpd.conf(5) man pages into more useful locations, so enter the command `make install`. Manually copy the sample vsftpd.conf file into /etc.

If you want to run vsftpd as a standalone dæmon, create a startup script for vsftpd in /etc/init.d. Otherwise, configure either inetd or xinetd to start it up as needed (see the Standalone Dæmon vs. inetd/xinetd section).

If you're running vsftpd as a standalone dæmon, enable the startup script with `chkconfig` if you use an RPM-based Linux distribution or with `update-`

`rc.d` if you run Debian GNU/Linux. Alternatively, if you install vsftpd from an RPM or deb package, all these steps are executed automatically, with the probable exception of the last one. Did I mention that binary packages are much more convenient? Some distributions require manual intervention to enable newly installed packages. For example, on my SuSE 9.0 system, although the SuSE vsftpd RPM automatically installed /etc/init.d/vsftpd, I had to issue the commands `chkconfig --add vsftpd` and `chkconfig --level 35 vsftpd on` to enable the script.

vsftpd's Documentation

Before I begin a discussion of vsftpd that focuses narrowly on running it as a standalone dæmon serving up only anonymous FTP, I should point out some valuable, much more complete, sources of vsftpd documentation. First, vsftpd comes with an EXAMPLE/ directory containing sample configurations for a variety of FTP scenarios, including running standalone, running with xinetd, serving anonymous users only and serving local users. If you installed vsftpd from source code, EXAMPLE is a subdirectory of your vsftpd source code tarball. If you installed vsftpd from a binary package, it's probably been copied to your system somewhere under /user/share/doc. It is /usr/share/doc/packages/vsftpd/EXAMPLE on SuSE systems.

As I mentioned in the previous section, vsftpd has man pages, vsftpd(8) and vsftpd.conf(5). Finally, the default (sample) vsftpd.conf file itself is well commented. It doesn't contain all vsftpd options, but it does illustrate the most commonly used ones. I've successfully gotten vstpd to work several times with only minimal tweaking to the sample vsftpd.conf file.

Standalone Dæmon vs. inetd/xinetd

Before configuring vsftpd itself, you must decide whether to run it as a standalone dæmon or by way of a super-server, inetd or xinetd. In previous versions of vsftpd, its developer recommended using it with xinetd due to xinetd's logging and access-control features. However, vsftpd versions 1.2 and later have native support for most of those features. For this reason, Evans now recommends that vsftpd be run as a standalone dæmon. In addition, a performance cost is associated with using inetd or xinetd. The cost isn't warranted if your system is to be a dedicated FTP server or if you anticipate FTP comprising a significant percentage of your system's activity.

I'm going to take the liberty of using standalone dæmon examples for the remainder of this article. vsftpd's included documentation amply describes how to use vsftpd with inetd and xinetd; see the example configurations included in vsftpd's EXAMPLE directory.

Interestingly, the vsftpd package that comes with SuSE 9 is preconfigured to run from xinetd, while Debian 3.0's runs from inetd. This is especially logical in the latter case, because Debian 3.0 comes with an older version of vsftpd (1.0.0), but SuSE 9.0 uses vsftpd 1.2. The vsftpd RPMs that come with Fedora and Red Hat install vsftpd as a standalone dæmon. At any rate, there are two steps to converting vsftpd from inet/xinetd startup to standalone startup.

First, as I mentioned in the Getting and Installing vsftpd section, you must make sure you've got an enabled startup script for vsftpd in /etc/init.d. The Fedora Core 1 and SuSE 9.0 packages both provide and install one; in SuSE's case it's present but disabled by default, in favor of xinetd. If you used Debian 3.0's vsftpd package or installed vsftpd from source, however, you need to create your own startup script. You also must create the corresponding links in the directories for the runlevels at which you want vsftpd to run, such as rc3.d and rc5.d. The last step is easy to do automatically with `chkconfig` or `update-rc.d`.

Second, you need either to disable vsftpd's xinetd file, by setting `disable = yes` in the file /etc/xinetd.d/vsftpd or to comment out vsftpd's line in /etc/inetd.conf. Alternatively, you can disable inetd or xinetd altogether, if vsftpd was the only important thing it was starting.

Arguably, it's irresponsible of me to recommend that you enable an application's startup script before you've fine-tuned that application's security. In my opinion, enabling is one thing; you're fine so long as you follow through and lock down the service before actually starting it or rebooting your system.

Third, you need to make sure that in /etc/vsftpd.conf the parameter listen is set to YES. This brings us to vsftpd configuration proper.

Configuring vsftpd for Anonymous FTP

Actually, you may not need to do anything more to configure vsftpd for secure anonymous FTP. Its default configuration settings permit only anonymous FTP. What's more, no write commands of any kind are enabled by default, and in recent versions of vsftpd, the dæmon chroots itself to the directory /usr/share/empty whenever possible. This is one of the things I love about vsftpd. It actually takes more work to loosen its security than it does to tighten it down.

Assuming your distribution hasn't altered this default behavior, all you need to do now is populate your anonymous FTP user account's home directory with FTP content for people to download. On Debian 3.0, SuSE 9.0 and Fedora Core 1, the anonymous FTP user is ftp by default, with a home directory of /srv/ftp for Debian and SuSE and /var/ftp in the case of Fedora. If you installed vsftpd from source, the anonymous FTP directory is whatever home directory you

assigned to the anonymous FTP user account you created. Pay special attention to ownership and permissions when populating your FTP directories. Defaults may or may not be appropriate, but at least do a quick `ls -al` now and then to see for yourself.

Even though default settings suffice for many users, let's take a closer look at the vsftpd.conf parameters most relevant to anonymous FTP. By default, this file resides in /etc, but on Red Hat and Fedora systems it resides in /etc/vsftpd/. Listing 1 shows a sample vsftpd.conf file.

Listing 1. vsftpd.conf Settings for Anonymous FTP

```
listen=YES
# listen_address=
anonymous_enable=YES
ftp_username=ftp
# anon_root=[$ftp_username's home directory]
write_enable=NO
anon_upload_enable=NO
anon_mkdir_write_enable=NO
anon_other_write_enable=NO
anon_world_readable_only=YES
anon_max_rate=0
idle_session_timeout=300
ascii_download_enable=NO
ascii_upload_enable=NO
connect_from_port_20=NO
port_enable=YES
hide_ids=NO
log_ftp_protocol=NO
syslog_enable=NO
max_per_ip=0
# cmds_allowed=
local_root=/usr/share/empty
nopriv_user=nobody
ftpd_banner=(vsFTPd 1.2.0)
```

In practice, you'd never use a vsftpd.conf file exactly like Listing 1. All parameters in it are, in fact, set to their default values. Rather, this listing is meant as a quick reference. Let's discuss its parameters in turn.

- listen: tells vsftpd to run as a dæmon rather than as a per-connection process invoked as needed by inetd or xinetd. Default value is NO.
- listen_address: specifies on which local IP address vsftpd should listen for connections. The default is `""` (null), signifying all local IP addresses. If you want to run multiple virtual FTP servers, you need to set this parameter in each virtual server's configuration file (see the next section, Virtual Servers).
- anonymous_enable: this parameter, whose default is YES, determines whether vsftpd accepts anonymous logins. If set to YES or not set at all, vsftpd accepts connections from the users anonymous and ftp (the two are equivalent) without requiring a real password.
- ftp_username: the name of the user account used for anonymous logins, that is, FTP logins as anonymous and ftp. This account must exist in /etc/

passwd and should have a valid home directory that is not owned by the user account; the default is ftp.

- anon_root: the directory vsftpd should chroot into for anonymous logins. This defaults to the home directory of the anonymous ftp user account (see ftp_username), but you can use this parameter to set a different anonymous FTP root. Either way, this directory should not be owned by the anonymous ftp user.
- write_enable: unless this parameter is set to YES, no user may upload any files under any circumstances, regardless of other settings in vsftpd.conf. Its default value is NO.
- anon_upload_enable: if this parameter and write_enable are both set to YES, anonymous users are permitted to upload files into directories on which the anonymous user account has write permission.
- anon_mkdir_write_enable: if this parameter and write_enable are both set to YES, anonymous users are permitted to create new directories within directories on which the anonymous user account has write permission.
- anon_other_write_enable: if this parameter and write_enable are both set to YES, anonymous users are permitted to delete and rename directories within directories on which the anonymous user account has write permission.
- anon_world_readable_only: if set to YES, this parameter forbids anonymous users from downloading any non-world-readable file. Most useful if anonymous users are able to upload files you don't want other anonymous users to download.
- anon_max_rate: specifies the maximum data transfer rate, in bytes per second, that anonymous users can use. The default value is 0, which means unlimited.
- idle_session_timeout: the maximum amount of time, in seconds, allowed to transpire between FTP commands until a session is closed forcibly by the server. Default value is 300, but if you're worried about denial-of-service attacks, you may want to set this lower.
- ascii_download_enable: if set to YES, this allows users to perform ASCII-mode downloads, as opposed to binary-mode. The default is NO because ASCII-mode is seldom if ever necessary, and it's much less efficient, so much so as to represent a potential vector for denial-of-service attacks.
- ascii_upload_enable: ASCII-mode uploads, on the other hand, are sometimes necessary for such things as scripts. This parameter's default value is, nonetheless, NO.
- connect_from_port_20: in active-mode FTP sessions, whenever a user downloads anything, including directory listings, the server initiates a new connection back to the client, conventionally originating from the server's TCP port 20. By default, however, vsftpd originates such connections from

a higher, non-privileged port, in order to avoid having to run as root. To change this default behavior, in case your FTP users connect from behind proxies or firewalls that don't expect such behavior, set this parameter to YES.

- port_enable: set this to NO to disable PORT commands, which effectively disables active-mode FTP altogether. Default is YES.
- hide_ids: if set to YES, replaces the owner and group fields in all directory listing output to ftp and ftp, respectively. Personally, I think this can be a useful bit of obscurity when used on public FTP servers, but the default is NO.
- log_ftp_protocol: if set to YES, turns on per-command logging, FTP protocol commands, that is, triggered by but distinct from FTP user-space commands. Invaluable for troubleshooting.
- syslog_enable: normally vsftpd writes log messages to /var/log/vsftpd.log. Setting this parameter to YES (its default is NO) sends those messages instead to the system's syslog service, using the FTPD facility.
- max_per_ip: specifies the maximum number of concurrent connections permitted from a single source IP address. Limiting this may seem like a good idea—the default is 0, which means unlimited—but doing so has a disproportionate effect on users connecting from behind NAT/SPAT firewalls, which cause multiple users to appear to originate from the same source IP address.
- cmds_allowed: specifies a comma-separated list of allowed FTP commands; default value is `""` (null), which means unlimited. Only FTP protocol-level commands may be specified, not the commands commonly accepted by FTP client software packages. For example, to allow clients only to list files, change working directories and download files, you'd use `cmds_allowed=USER,LIST,NLST,CWD,RETR,PORT,QUIT`. The Web site www.nsftools.com/tips/RawFTP.htm is a useful reference for these commands.
- local_root: this specifies an empty, root-owned directory to which vsftpd chroots itself any time it doesn't need access to other parts of the filesystem. Default value is /usr/share/empty.
- nopriv_user: specifies the non-privileged user vsftpd runs as whenever possible. vsftpd obviously needs to be root when doing things like binding to TCP port 21. It demotes itself as soon as it can, however, in order to lessen the chance of a buffer-overflow vulnerability or other process-hijacking event leading to root compromise.
- ftpd_banner: banner message to display when FTP clients attempt to connect. Default message is hard-coded into vsftpd; in v1.2.0, it's simply (vsFTPd 1.2.0). Alternatively, you can use the parameter banner_file to specify a text file containing your banner message.

The vsftpd.conf(5) man page explains these and many other parameters you can use. Believe it or not, I've only scratched the surface here.

Virtual Servers

If you want to have multiple virtual FTP servers residing on the same physical host, one with multiple IP addresses, vsftpd can do this easily. All you need to do is run multiple instances of the vsftpd dæmon, each with its own vsftpd.conf file specifying on which IP address to listen and which directory to use as its anonymous root.

For example, suppose I've got two IP addresses assigned to my machine, 1.2.3.4 and 1.2.3.5, registered in DNS to the names knusper and rover, respectively. In that case, I could have two configuration files for vsftpd, say, /etc/vsftpd.knusper and /etc/vsftpd.rover. Listings 2 and 3 show these files.

Listing 2. Virtual FTP Server Configuration File /etc/vsftpd.knusper

```
listen=YES
listen_on=1.2.3.4
connect_from_port_20=YES
anonymous_enable=YES
anon_root=/srv/ftp/knusper
ftpd_banner=Welcome to FTP at knusper.wiremonkeys.org. Behave!
```

Listing 3. Virtual FTP Server Configuration File /etc/vsftpd.rover

```
listen=YES
listen_on=1.2.3.5
connect_from_port_20=YES
anonymous_enable=NO
ftpd_banner=Private FTP at rover.wiremonkeys.org. Strangers-B-gone.
# DANGER: don't use the following unless you know what you're doing!
local_enable=YES
```

Notice my possibly foolish use of the local_enable parameter in Listing 3. It's dangerous to set this to YES, because FTP logon credentials are sent in clear text. You never want to expose real system credentials to eavesdropping, especially if your server is Internet-connected. The real reason I show it here is to illustrate that because each virtual server uses its own configuration file, you can specify completely different behaviors for each. One virtual server may have a public uploads directory that anonymous users write to, whereas another may be a strictly read-only FTP site. Conversely, you need to take care that settings you consider to be important in preserving overall system security are set consistently between different virtual servers running on the same machine.

Besides creating different configuration files for each virtual FTP server you want vsftpd to serve up, you also need to alter your startup script accordingly.

The startup script on my sample server, represented by Listings 2 and 3, would need something equivalent to these two lines:

```
vsftpd /etc/vsftpd.knusper
vsftpd /etc/vsftpd.rover
```

If you run Red Hat or Fedora, this already has been taken care of for you. The /etc/init.d/vsftpd script included with those distributions' vsftpd RPM packages automatically parses the directory /etc/vsftpd for as many configuration files as you care to put there, so long as the filename of each ends with .conf. This strikes me as an excellent bit of foresight on the part of the Red Hat team.

That's all you need to know about setting up a simple and secure anonymous FTP server with vsftpd. As I mentioned, I've only covered a subset of what vsftpd is capable of doing. Despite its minimalist design philosophy, this is a powerful FTP server indeed. Fortunately, it's also well documented, so it's really no cop-out for me to refer you to the vsftpd.conf(5) man page and the EXAMPLE/ directory for information on the many other uses of vsftpd.

Mick Bauer, CISSP, is *Linux Journal*'s security editor and an IS security consultant in Minneapolis, Minnesota. He's the author of *Building Secure Servers With Linux* (O'Reilly & Associates, 2002).

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

EOF

Carrier Grade Linux

Ibrahim Haddad

Issue #123, July 2004

Your next mobile phone server will run Linux. Leading telecom vendors are enhancing Linux for reliability, high availability, scalability and more.

In January 2002, the Open Source Development Labs (OSDL, www.osdl.org) established the Carrier Grade Linux (CGL) Working Group. This initiative was intended to enhance the Linux kernel to achieve a highly available, secure, scalable and easily maintained open-source platform suitable for carrier grade systems.

Many companies joined the CGL initiative, and today the CGL is composed of member companies that work together contributing to the CGL requirement definition, helping current and starting new open-source projects to meet these requirements. Many CGL member companies already have contributed pieces of various technologies to open source to make the Linux kernel a more viable option for telecom platforms. CGL activities are providing momentum for Linux in the telecom space, allowing it to be an alternative to proprietary operating systems.

Gateways, signaling and management servers are the three main areas into which the CGL Working Group expects the majority of applications implemented on CGL platforms to fall. In addition to specifying the requirements, the Working Group also identifies existing open-source projects supporting the road map, implementing required components and interfaces of the platform. When an open-source project does not exist to support a CGL requirement, the Working Group launches or supports new projects to implement the missing functionality.

The scope of the CGL Working Group covers two main areas, carrier grade enhancements to the kernel and development tools. Kernel enhancements

cover availability, security, scalability and reliability, as well as changes to interfaces for hardware, user-level code, application code and development and debugging tools. Software development tools covered by CGL include debuggers and analyzers.

The CGL Requirements Definition Version 2.0, released October 9, 2003, divides the requirements into the following main categories:

1. Clustering supports the use of multiple carrier server systems providing higher levels of service availability through redundant resources and recovery capabilities.
2. The security requirements aim at maintaining a certain level of security while not endangering the goals of high availability, performance and scalability. These requirements support the use of additional security mechanisms to protect the systems and provide special mechanisms at kernel level to be used by telecom applications.
3. Standards: CGL specifies standards to which compliance is required, including the Linux Standard Base, POSIX standards and a number of Internet RFCs.
4. CGL specifies platform requirements that support interactions with the hardware making up carrier grade systems. Examples of platform requirements include: hot insert, hot remove, remote boot support, boot cycle detection and support for diskless systems.
5. Availability requirements support heightened availability of carrier grade systems, such as improving the robustness of software components or by supporting recovery from failure of hardware or software. Examples include support for watchdog timer interface, disk and volume management, Ethernet link aggregation and link failover and application heartbeat monitor.
6. Serviceability requirements support the availability of applications and the operating system. Examples include support for producing and storing kernel dumps, dynamic debug of the kernel and running applications, platform signal handler and remote access to event logs.
7. Performance requirements support performance levels necessary for the environments a carrier grade system would encounter. Examples include support for application (pre) loading, soft real-time performance, kernel preemption and RAID 0 support.
8. Scalability requirements support vertical and horizontal scaling of carrier server systems, such as the addition of hardware resources to result in acceptable increases in capacity and throughput.
9. Tools requirements provide capabilities to facilitate diagnosis, such as the support for a kernel debugger, kernel dump analysis and the capability to debug multi-threaded programs.

Many individuals within the CGL initiative are active participants in the mainline Linux development community. In addition, the implementations providing the carrier grade enhancements to the kernel are open-source projects and are planned for integration with the Linux kernel. All of the enhancements are available from their respective project Web sites; please refer to the OSDL Web site for links.

As of January 2004, the CGL Working Group is developing CGL version 3.0. The group expects to release the final official version by October 2004. The participation in OSDL CGL is open to everyone. For more information, please visit the OSDL Web site.

Ibrahim Haddad, contributing editor to *LJ*, is a Researcher at the Ericsson Research & Innovation Department in Montréal, Canada.

Archive Index Issue Table of Contents

Advanced search

Arkeia 5.2 Network Backup

Dan Wilder

Issue #123, July 2004

The design shows careful thought about what a backup manager needs.

Product Information.

- Vendor: Arkeia
- URL: www.arkeia.com
- Price: $590–$1,190 US for three to seven computers; larger systems may range up to $20,000 US depending on configuration.

The Good.

- Multiple platform.
- Centrally scheduled backups.
- Hot-backup plugins.
- Browseable index of all tapes.
- Good documentation.

The Bad.

- Non-standard GUI.
- Disappearing error messages.
- Incomplete context-sensitive help.

We haven't checked on Arkeia since April 1999 (/article/3166), so we thought we'd take another look and see how this software is coming along.

Features

Arkeia Network Backup is a heterogeneous network client/server backup solution using a Linux or UNIX backup server. Client system backup software is available for Linux, as well as a variety of UNIX and UNIX-like OSes, including Mac OS X and Microsoft Windows 98, ME, NT, 2003 and XP.

Plugins are available for hot backup of applications including Oracle, Microsoft Exchange, Lotus Notes, IBM DB2 and MySQL.

Supported backup media include popular SCSI tape drives, libraries and autoloaders.

Arkeia Disaster Recovery, a separate product not reviewed here, provides bare-metal recovery for backed-up Linux clients and servers. Both Network Backup and Disaster Recovery are available for free 30-day demos. A third product, Arkeia Lite, suitable for backing up one Linux server and two desktop systems, is available at no charge.

We reviewed Arkeia 5.2.7 Network Backup, downloaded from www.arkeia.com, along with PDF documentation. The Linux version supports Debian GNU/Linux 2.2 and 3.0, Mandrake 7.2–9.2, Red Hat 6.0–9.0, Slackware 8.0 and SuSE 7.1–9.0.

Installation

The documentation, downloaded as a PDF, had about 500 pages of material, enough to be a little intimidating. The shortest document was the Quick Start Guide. I began there.

My bench system's distribution of the day was Debian 3.0. The Debian installation for Arkeia came as a .tar.gz file, not as a Debian package. I unpacked this, cd'd to the top-level directory and then ran `install`, accepting all defaults.

Next I started xarkeia. Its futuristic design, as shown in Figure 1, takes some getting used to.

Figure 1. Option screen from the Savepacks Menu showing icons, in the bar above advanced options, for navigating back up to the root menu.

Continuing with the Guide's instructions, I set a password for the Arkeia root user and configured and ran a dummy backup. As long as I followed the directions carefully, all went as indicated. At one point I did something out of sequence, attempting to start a backup prior to configuring any tapes. The backup stalled, and I was unable to configure tapes or to abort the backup using the GUI or anything else I knew to do at the time. A note on the support Web site elicited an e-mail response within a half hour, telling me to stop and restart a dæmon. I was then able to proceed. The dummy backup ran without further incident, and initial installation was complete. Arkeia was ready to configure real backups.

Architecture

Arkeia organizes things using a database where the administrator sets up:

- Tape drives.
- Drivepacks: groups of similar tape drives.
- Tapes: each with its own label and history information.

- Tape pools: groups of similar tapes.
- Savepacks: groups of files and directories backed up together.
- Backups: a backup uses one drivepack to store one savepack in tapes from one tape pool.
- Users: a variety of user roles are available, allowing the work of managing backups at a large site to be delegated.
- Servers: one installation may extend across multiple backup servers.
- Clients: multiple client systems are accommodated per server.

Backups are controlled and scheduled from the backup server. Backups can be manual or automatic, called Periodic by Arkeia. They also can be complete or incremental, an arrangement by which files that have not changed since a baseline are not backed up. Incremental backups are scheduled in a multilevel fashion, with the baseline for a given level being the previous lower-level backup. Backup level is the same for all files in a given periodic backup.

You can schedule any backup to put multiple backups on one tape, filling the tape, or you can start a new tape.

A savepack contains items to be backed up, such as files, databases and directories. One savepack can contain items from multiple hosts. An item can be backed up using a plugin, such as the one for MySQL.

Libraries, stackers and so on have special management interfaces under Arkeia but are configured as sets of drives enrolled in drivepacks. From the point of view of managing a backup itself within Arkeia, there's not much difference between a library and any other collection of drives.

The GUI

Arkeia can be managed through a GUI client program, xarkeia, or by means of a set of command-line clients. The actual work is done by dæmons reached through these clients. Many system administrators may find the GUI easier to use for routine operations and configuration.

The xarkeia GUI was written from scratch on top of X, using no Motif, Qt, GTK or any other third-party GUI libraries. The appearance is distinct and different, and although it requires some getting used to, was easy to use after only a little practice. The window decoration buttons you might be used to seeing along the top bar aren't there; they are replaced by a circle of buttons in the upper-left corner. I missed the ability to move an instance of xarkeia from one virtual desktop to another.

xarkeia has an error message panel near the top that was a source of some annoyance. It featured error messages that vanished too quickly for careful reading.

Context-sensitive help is provided from a Help button in the button circle. Based on an unscientific sampling, I found meaningful help messages in only about half of the screens. There is room for some improvement here. Experienced system administrators should not shy away from reading manuals, however, and I found the Arkeia User's Guide to be complete and comprehensive.

I was unable to discover much by way of customization available for the GUI. The colors and fonts it comes with are, as far as I can tell, the ones you will live with.

Other than that, I had no problems with xarkeia. Among the many features I liked was what they called their function path bar. If you use many applications having multiple levels of menus, you're no doubt all too used to clicking back, back and so on until you climb back out to the top-level menu. xarkeia's function path bar, as shown in Figure 1, stacks the icons you've used as you descend to the lower levels of a menu tree. Clicking an icon in this bar can take you back out through multiple menu levels with only a single mouse click.

A Real Backup and Restore

The Arkeia User's Manual is the next stop after finishing the Quick Start Guide. At 330 pages, there's a lot of reading there. I opened the User's Manual using xpdf and continued on to configure some real backups using an Exabyte VXA drive. The drive was detected and configured easily. A new drivepack was defined, several tapes entered and labeled and enrolled in a pool. The tape labeling dialog could have used an eject button. I already had a savepack set up from the Quick Start Guide exercises, so I ran an interactive backup and then configured a periodic one to run several times, allowing me time to add and delete files in between.

Within the Restoration menu, files can be selected by filename search or by file tree browser. The “Invalid regular expression!” popup I got when I clicked Search puzzled me, until I read the related section of the User Manual, which pointed out that I had to check some boxes in the search screen as well as enter search words into adjacent text boxes. “You must check at least one checkbox” would have been more helpful.

Restoration offered many options with respect to where to restore, ownership, access rights, overwrite of existing files, verifying backed-up files and so on. After selecting files using the backup browser, I started restoration only to be

told “Please insert tape Monthly22” in one tape drive. After a little bit of guessing I was able to initiate restoration.

Tape Indexing

Arkeia maintains on-line tape index, history and configuration information that it uses when it is time to restore. This index makes it possible to browse the backed-up date on-line for easy restoration. The downside, as always with such an arrangement, is that there is a point of failure: loss of the on-line index.

This index is kept in the Arkeia installation directory, by default /opt/arkeia, in the server/dbase subdirectory. In the event of disaster, it can be reconstructed from the tapes by use of provided utilities. Every backup tape must be fed through, which can be a lengthy and laborious process. The fact that index rebuilding is supported at all is a good thing. I've used backup products with on-line indexes but no way to rebuild them from tape. Having a way to rebuild these if need be is good; however, it is better to avoid being in a place where this is needed.

Arkeia Disaster Recovery provides facilities to handle this situation, accommodating bare-metal restore of the backup server or any of the client systems direct from tape. For the prudent but more adventurous administrator, who might want to restore using a standard installation method, followed by bringing in backed-up data files from tape, Arkeia Support advises me that an up-to-date copy of the arkeia install directory kept in a safe location, augmented by snapshots of the server/dbase directory taken after each backup, should suffice to allow a restore even following a loss of the backup server. Always test your restoration procedure. Your results may vary.

Other

The product accommodates tape duplication to use, for example, if you want to keep both on-site and off-site copies of backup tapes.

The command-line clients, covered partly in the ending chapters of the User's Guide, partly in the 137-page Command Line Interface Manual, permit you to manage Arkeia through a command-line interface and let you access this software by way of your own scripts. Some sample scripts are provided.

Overall

I like this product. The GUI takes a little getting used to, and the fit and finish is rough in some places. The design shows careful thought about what a backup manager needs. Cross-platform support is good. The command-line clients ensure that you're not stuck with a closed-in GUI that won't let you get beyond

what the GUI provides. The Quick Start Guide offers a painless introduction to the software, and the other two manuals pick up nicely where it leaves off. Free 30-day evaluation support was quick and courteous.

Dan Wilder is technical manager at Specialized Systems Consultants, Inc.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

From the Editor

Traps? Who Needs Them?

Don Marti

Issue #123, July 2004

Use the development tools in this issue to clear the way for a low-pain migration from your remaining non-Linux systems.

Our reader surveys consistently show that a lot of our readers are still running, and even developing code for, non-Linux operating systems. Whatever system you're coding for today, you can use Linux as your development platform and give yourself the flexibility of moving to Linux tomorrow.

In a recent essay entitled “Free But Shackled—The Java Trap”, Richard Stallman wrote, “If you develop a Java program on Sun's Java platform, you are liable to use Sun-only features without even noticing. By the time you find this out, you may have been using them for months, and redoing the work could take more months.” That's bad news, but platform lock-in isn't only for Java developers. So build and run your projects on an all-free system regularly to save yourself from accidental lock-in as you work. We're happy to offer four complete cross-platform sample applications in this issue.

First, Python is one of our favorite languages here at *Linux Journal*. Its simplicity makes old code, and other people's code, easy to understand and maintain. Let David Reed's GladeGen (page 40) write the GUI code for you, so you can focus on business logic.

The best development frameworks are the ones that grow cooperatively with an application that depends on them. Mozilla is a good example. Don't get Mozilla the browser mixed up with Mozilla the framework—work through Nigel McFarlane's process viewer utility on page 66 to understand how to use Mozilla's framework to write non-Web applications.

As dedicated followers of fashion, we're happy to have a blog tool written with the .NET-compatible Mono. Ian Pointer shows how it's done and scores two buzzword points on page 50.

If you need to develop for Microsoft Windows, you can save yourself the expense and learning time of another set of development tools. Joey Bernard's article on MinGW (page 58) shows you how to add Windows support to your Linux apps or bring Windows apps to Linux with minimal rewriting.

Finally, on page 83, John Healy, Andrew Haley and Tom Tromey explain how Red Hat made the popular Eclipse integrated development environment build natively with no Java Virtual Machine (JVM), and no proprietary dependencies, at all. Maybe Java developers are finally getting the hang of this cross-platform thing. Stay out of traps, don't fall for lock-in, and enjoy the issue.

Don Marti is editor in chief of *Linux Journal*.

Archive Index Issue Table of Contents

Advanced search

Readers sound off.

No Accounts for the Penguins?

On a recent trip to the Camden Aquarium in Camden, New Jersey, I found that the penguins were attracted to my Linux hat. My girls loved the penguins, and I plan on raising them on Linux desktops (the girls that is).

—
Shane M. Parker

BSD, Please

May I suggest including other open-source UNIX-like operating systems? I have just installed OpenBSD for a project at work. Although it would have been easier with Linux, if the boss says BSD then it is BSD. I must admit that it was a pain; however, after going through the process for the right application environment, OpenBSD would be a good choice. In my opinion, Lintel is no better than Wintel. I would urge you to consider articles on ports, the BSD solution to software distribution. Two choices are better than one but not much better.

—
raymond a jacob jr

Check out www.lnx-bbc.org/garticle.html for an updated version of the *Embedded Linux Journal* article on GAR, a ports-like software build system. —Ed.

LJ Web Site—Too Political?

The article I received via e-mail titled “Hot Air” (linuxjournal.com/article/7493) was a huge disappointment to me. Now, I am not a conservative. I am a liberal, but not the socialist flavor represented by Kerry and Kennedy. This, however, is all beside the point. Using this technical forum to promote these views is the kind of thing that frankly makes me want to cancel my subscription. Having said that, I'm willing to give *LJ* one more chance, but I will be paying closer attention to what comes in my e-mail. When I want politics, I know where to go to get it, and I won't support your magazine with my dollars so they can be spent promoting political agendas.

—
Jake Fear

Doc Searls replies: if Air America were a right-wing network in the same position, I would have covered it the same way. Was the whole thing a stretch as a Linux subject? Sure. But I think it would help broadcasters of all sorts to take advantage of Linux, the Net and technologists who understand both. That's what I was trying to get across.

We've been writing about the intersection of Linux, the Net and radio for years, by the way. The Web site has much more room for material, and much more leeway around subject matter, than the magazine.

I'm sorry the piece disappointed you, and that you saw it promoting a political agenda. For what it's worth, I'm a registered independent. In recent elections I've mostly voted Libertarian. If there's an ax I try not to grind, that's the one.

Thanks for the Recommendation

Wow! After reading Marcel Gagné's article on SuperKaramba [*LJ*, May 2004], I took just a few minutes and configured it for my system. Hats off to all the SuperKaramba and theme authors and to Marcel for bringing this to your readers' attention. Thanks for the great magazine and the great articles.

—
Robert Fields

Everybody SPF Now!

Why would your publication be so pro-SPF without publishing an SPF record for your own domain?

—
Nicholas Marsh

Try it now, with `host -t txt linuxjournal.com`. More SPF info at spf.pobox.com. —Ed.

Shielded CPUs vs. RTLinux/RTAI

I enjoyed the article entitled “Shielded CPUs: Real-Time Performance in Standard Linux” in the May 2004 issue of your magazine. I have two brief comments about this article that should be perceived as constructive criticism.

1) Steve Brosky could have been even more informative in his performance numbers had he also compared the interrupt response times of his shielded CPUs technique to the more conventional RTLinux/RTAI (which he himself acknowledges as being a very good standard by which to judge a real-time solution). The advantage of providing actual numbers that compare his system to RTLinux/RTAI is that it would have helped people to decide more quickly which RTOS is right for them.

Specifically, RTLinux and RTAI have better performance numbers, with latency and jitter measured in the 10s of microseconds, not 100s, as was demonstrated using shielded CPUs. For some applications the shielded CPUs technique is sufficient, while for others it is not. Having had sets of numbers representing RTLinux/RTAI would have helped people to decide more quickly which real-time approach is right for them.

2) The article should have spent more time discussing the potential pitfalls inherent to the shielded CPUs technique. Specifically, CPU shielding still suffers from priority inversion problems inherent in any system that shares resources with non-real-time components. In order for CPU shielding to guarantee hard real-time performance, one must be careful to write programs that do not access any potentially blocking operations in the Linux kernel.

More specifically, any swapping of memory to disk in the shielded process throws real-time performance out the window. One has to do things like lock all process memory to RAM, for example, mlockall(), as well as be careful not to access any device drivers that may run on non-shielded CPUs or that may sleep. This must be done in order to avoid priority inversion with the normal, non-real-time, Linux system.

Overall, a good article, but it could have been even more informative, in my opinion, had the two points above been addressed.

—
Calin A. Culianu

Steve Brosky replies: there are a lot of things that could have gone into the article—but with limited space you pick and choose what to include.

Photo of the Month: More Fashion Ideas

Robert Henry's Halloween picture in the May 2004 issue of *Linux Journal* prompts me to send you this photo of my two children taken two years ago.

—
Dr Stuart DeGraaf

Photo of the Month gets you a one-year extension for your subscription. Anyone have a hobby other than sewing? Photos to info@linuxjournal.com. —Ed.

Bank Answer One: Monoculture Is Risk

I just read [William Mitchell's] letter to the editor in the May 2004 issue about Microsoft IE-only bank Web sites. As he mentioned that his bank did this, a letter to the chairman of the board and the board of directors might do the trick. I suspect if one points out that mandating usage of the least secure Web browser is not only an annoyance to those who use more secure ones, but that insisting on this puts the bank at risk for more theft, this may change.

—
Anonymous

Answer Two: Moo-ve to Another Bank

What I would do in Mr Mitchell's case is inform my bank that *I* am the customer. They exist to service *me*, and not like a bull services a cow, either. If they choose not to do that, then I will simply find a bank that will.

—
John McKown

Answer Three: Fake It

I've been using User Agent Switcher for Mozilla and Firefox to get to sites that claim to be IE/Netscape-only friendly. It's been working fine. You can check it out here: www.chrispederick.com/work/firefox/useragentswitcher.

I'd e-mail William Mitchell but you don't post e-mails in the Letters section, for obvious reasons! Maybe you could pass this on to him.

—
Sam Mingolelli

Another Satisfied Reader

I just wanted to let you know how much we all look forward to reading your magazine each month. Keep on doing what you're doing.

—
Dan

sudo adduser bhm

We've run enough photos of readers' and contributors' children that it's time for one from an editor. Here's a photo of my son Bilal at age zero days.

—
Don Marti

Archive Index Issue Table of Contents

Advanced search

UpFront

diff -u: What's New in Kernel Development

Zack Brown

Issue #123, July 2004

Roland Dreier and the folks at OpenIB.org have produced a rough cut of an **InfiniBand** stack, including a low-level driver for **Mellanox HCA** hardware; upper-layer protocols such as IP-over-InfiniBand, SCSI RDMA protocol, sockets direct protocol (SDP), uDAPL and MPI; and accompanying user-space utilities. The code itself is open source, but Microsoft has intellectual property claims on SDP and does not automatically allow it to be used in open-source projects. As a result, Roland and the others have split their InfiniBand stack into free and encumbered packages, a decision that seems to satisfy everybody for the moment.

Intel has started a **SourceForge** project for its **PRO/Wireless 2100** miniPCI network adapter driver for Linux kernels in the 2.4 and 2.6 series. Although the firmware is binary-only, the rest of the project seems to be in keeping with open-source methods. A public mailing list serves to connect developers, and new updates are published frequently, so everyone can try them out and report problems or suggest enhancements. Currently they've classified the code as early beta, so various bugs and missing features should be expected. At the same time, the Intel developers intend to be particularly sensitive to issues arising from interactions with particular Linux distributions, so hopefully the default installations of most distributions will break in only known, documented ways.

Niraj Kumar has ported **UFS1** and **UFS2** to Linux. UFS1 has been the native BSD filesystem for a long time, and UFS2 is a recent extension, adding such features

as 64-bit block pointers and extended file storage. Niraj's Linux port is read-only at the moment, as work has just begun. UFS2 itself is quite new, and even on the BSD operating systems it does not yet support such things as the GRUB bootloader. It is the default only on FreeBSD systems; NetBSD still creates a normal FFS filesystem by default. Originally derived from UFS1 by **Kirk McKusick** and **Poul-Henning Kamp**, UFS2 is undergoing active development. Linux support apparently will follow close upon its adoption by the BSDs.

Michael Geng has produced a GPLed device driver for the I2C-based Videotext/Teletext decoder **SAA5246A**, providing the same interface as that of the SAA5249 chip driver. Based in part on work by **Martin Buck**, Michael has cleaned up the existing code and completed the work to the point that **Andrew Morton** has accepted it into the 2.6 tree as part of the official kernel sources. As Michael points out, newer TV cards no longer include these Teletex decoders and instead rely on the CPU to perform the same function. When present, however, these chips appear to do a better job and are worth supporting where possible.

Emulex has decided to open source the driver for its **LightPulse** Fibre Channel Adapter family and has created a SourceForge project page to accomplish this. The hope is to get the code cleaned up and completed and have it accepted into the 2.6 kernel tree. When a company decides to free up the source code for one of its drivers, they typically are given a warm batch of kudos from the kernel developers, as well as comments, criticism and patches from developers looking over the code for the first time. In this case, **Jeff Garzik** has done the most probing analysis, offering a ton of feedback to the Emulex developers. Apparently there are some ugly bits in the code, as the Emulex folks had warned in the announcement, but Emulex seems committed to doing whatever cleanup is necessary to make the driver acceptable to Andrew and the rest of the kernel folks.

Kristian Soerensen has been working recently on **Umbrella**, a new security project for handheld devices intended to help protect them from viruses and other cracks. One of the main features of Umbrella is its unambiguous configuration system. All complexity has been eliminated, so it is not possible for the user to mistakenly allow undesired breaches.

LJ Index—July 2004

- 1. Trillions of icy objects in the Oort cloud surrounding the solar system: 6
- 2. Transactions per minute (tpmC) of Oracle Database 10g on an NEC Intel system running SuSE Linux Enterprise Server 9: 609,467
- 3. Price-performance ratio in $/tpmC of the above: 6.78

- 4. Size in billions of dollars of the embedded software market in 2002: . 675
- 5. Projected size in billions of dollars of the embedded software market in 2007: 1
- 6. Mandrake percentage in DesktopLinux's 2004 Desktop Linux survey: 20.3
- 7. Red Hat percentage in DesktopLinux's 2004 Desktop Linux survey: 19.3
- 8. SuSE percentage in DesktopLinux's 2004 Desktop Linux survey: 16.0
- 9. Debian percentage in DesktopLinux's 2004 Desktop Linux survey: 11.1

- 1: Solarviews.com
- 2, 3: Oracle
- 4, 5: Gartner, in *BusinessWeek*
- 6–9: DesktopLinux.com

They Said It

The technology (software) industry really only has 75 leaders, they just continually recycle in and out of successful and failed companies.

—Eric Norlin (ericnorlin.typepad.com/weblog/2004/03/cracking_the_in.html)

When you experience bad service it's because that service is hostage to a business plan.

—Britt Blaser, telephone call

The most important operating system you write applications for ain't Windows, or Macintosh, or Linux. It's Homo Sapiens Version 1.0. It shipped about a hundred thousand years ago, there's no update in sight; but it's the one that runs everything.

—Bill Hill (channel9.msdn.com)

Open source has no secrets.

—Doc Searls

When you stop to think about it, you keep secrets from people when you don't want them to know the truth. Secrets, even when legitimate and necessary, as in genuine national-security cases, are what you might call passive lies.

—Walter Cronkite (www.linuxjournal.com/article/5031 and staugustine.com/stories/040404/opi_2233122.shtml)

I figured yesterday I might as well see how hard it would be to turn GtkTextView into a real editor.

I started with the three hundred lines of Python and a fairly simple Glade file I made on Christmas day. After two days of hacking, I have added a thousand lines of code, and now this thing is fairly usable. Yes, I'm now writing this log with the new editor.

—Lars Wirzenius (liw.iki.fi)

Archive Index Issue Table of Contents

Advanced search

Best of Technical Support

Our experts answer your technical questions.

Adding a New Service

I have written an SMS system in Java. I want to execute that system as a background service in Linux. My system is in a *.jar file. How can I do it as a service in Red Hat?

—
Kasun Perera

kasun@teamwork.lk

You need to create a script file that you would put in the /etc/rc.d/init.d/ directory. It must have a very specific format, as clearly indicated on this page: www.sensi.org/~alec/unix/redhat/sysvinit.html. I suggest that you look at other scripts in that directory to grasp the general format of the file, especially the first 15 lines or so.

—
Felipe Barousse Boué

fbarousse@piensa.com

When starting a new service, I like to copy the init script for SSH, because it's usually the simplest. Put in whatever commands you need to run to start your program from the command line. For example, you might need to set some environment variables to run a Java program. Run your init script from the command line to make sure it starts and stops your new service correctly, then use your distribution's tool for managing runlevels to make it run at startup. Use chkconfig on Red Hat.

—
Don Marti

info@linuxjournal.com

Speeding Up Webmin

I use Webmin and ZoneMinder on my P4 2.2GHz, 1GB RAM system, and the response is less than what I would expect. Is there a way to speed up the loopback device?

—
Howard Watts

howardwatts@sbcglobal.net

I just upgraded some Webmin systems to the latest Webmin, 1.140 at the time of this writing, and I noticed a substantial speed improvement in operation. I upgraded all modules as well. All this was done directly from within Webmin.

—
Felipe Barousse Boué

fbarousse@piensa.com

The only lag on the loopback device is traversing the TCP stack; this should be very fast. If you see performance issues, you may want to look at a program called top (see man top) to see if there are any out-of-control processes.

—
Christopher Wingert

cwingert@qualcomm.com

It is highly unlikely that the loopback device is the culprit. It's more probable that there's something else causing the performance lag, network function or otherwise. Perhaps it's having timeout issues due to failed DNS lookups or something similar?

—
Timothy Hamlin

thamlin@zeus.nmt.edu

Winmodem: Hack or Replace?

I can't get my modem to work. I have a Creative Labs Blaster v92 PCI internal modem. Linux recognized a Conexant chipset and attempted to install the driver, but I received an error message. Should I try installing SuSE Pro 9.0 instead?

—
Manny

manuel61@joimail.com

Probably the easiest and most advisable solution is to purchase a very inexpensive modem that is not a Winmodem. You probably would get a better setup with less complex stuff, and you will get a modem that will last for many future Linux generations. Besides, you will show manufacturers that we all want standard modems, not proprietary ones.

—
Felipe Barousse Boué

fbarousse@piensa.com

It's almost never necessary to upgrade your entire operating system simply to support a device. Without knowing the exact chipset you are using, I can suggest only that you first determine if yours is a full hardware modem or a so-called Winmodem. I suspect it is the latter, because if it were a full hardware modem you probably wouldn't be having any issues. Rather than updating your Linux distribution, figure out which driver it was attempting to load and then go to the Web to find an updated driver. Winmodem support under Linux is ever-increasing (see www.linmodems.org for a good place to start). If you are lucky, you will be able to find a newer, working, version of the driver you are looking for—and save yourself a lot of trouble to boot.

—
Timothy Hamlin

thamlin@zeus.nmt.edu

Are you looking for a tweaky project to help you understand Winmodems, or do you simply want a Net connection? Have a goal in mind before you decide between the above two answers, and remember that if you upgrade, you may need to redo your modem setup.

—
Don Marti

info@linuxjournal.com

Getting Started

I would like to learn Red Hat 9. Can I install Red Hat 9 software and Windows 2003 server (Beta) software on the same PC? My PC is a Dell PIII with 700MZ, 6GB hard drive and 128MB of RAM. I know I will need to partition the hard drive. I saw the software for sale on Amazon.com for about $70 US. I am new to this and am trying to learn, so any assistance is appreciated.

—
Bill

whitesock95829@yahoo.com

Yes, you can install Linux and Windows on the same machine to create what is called a dual-boot system. There are some details to watch, though. Red Hat 9 has been discontinued so, to play with Linux and and learn, I would get Fedora Core 1 (Red Hat-sponsored) instead. You can download it from fedora.redhat.com.

—
Felipe Barousse Boué

fbarousse@piensa.com

The easiest way to try Linux is download Knoppix from knoppix.org. This allows you to try Linux and not impact your hard drive. Most distributions can be downloaded for free, including Red Hat 9. You may want to search for a more up-to-date distribution such as Fedora Core 2.

—
Christopher Wingert

cwingert@qualcomm.com

I haven't personally tried setting up a boot manager with Windows 2003 server and don't intend to, but I have heard of several people doing so successfully. The procedure seems the same as with earlier versions of Windows. Here are the main things you need to be aware of:

1) Knoppix includes QtParted, a free software partition editor. The interface is not as polished, but QtParted does just as good a job of partition editing as PowerQuest's PartitionMagic and other proprietary programs, and you might as well get in the spirit by doing the job with free software. Knoppix, incidentally, makes a great rescue disk.

2) You'll want at least one partition for Linux and another of 125MB for a swap partition. Opinions vary about how to partition, with some people favoring putting the /home, /var and other directories on separate partitions. However, because you're just starting out, perhaps you want to use only one.

3) You might also want a FAT32 partition so that you can share files between your operating systems.

4) If Windows isn't already installed, install it first, right after you partition the drive. Windows does not tolerate another operating system during installation. You can work around the problems, but it's easier just to avoid them altogether.

5) When you're installing, make sure you install the GRUB boot manager. The installation automatically detects the presence of Windows, and the boot manager loads when the machine starts and offers you a menu for choosing which operating system you want to start.

—
Bruce Byfield

bbyfield@axionet.com

You don't need to resize your existing partitions with a partition editor if you're installing from scratch. All Linux distributions include a basic partitioning tool. If you do use a partition editor, keep in mind that if it fails you may lose important data. Make sure to back up your existing system first, and check that the backup is good before resizing any partition. Alternatively, as Rick Moen

suggests, once you have a backup, you might as well just restore it to new partitions and save yourself the partition resizing step entirely.

But, your 6GB drive is too small to run two current operating systems comfortably. You might want to add a new, larger drive for Linux.

More advice for new users, including why dual boot is usually a bad idea, is in "Welcome to Linux, 2004" on the *Linux Journal* Web site (www.linuxjournal.com/article/7516).

—
Don Marti

info@linuxjournal.com

Archive Index Issue Table of Contents

Advanced search

New Products

XE-800 Embedded PC, PathScale Compiler IKO Compiler Suite, gumstix Boards and Computers and more.

XE-800 Embedded PC

Octagon Systems released the XE-800, an SBC using the EPIC (embedded platform for industrial computing) form factor. Sized midway between the PC/104 and EBX form factors, the EPIC-based XE-800 is designed for embedded military, security, industrial and mobile applications. It can operate over a –40° to 75°C temperature range and features four USB 2.0 ports, two USB 1.1 ports, two eight-wire serial ports, 48 lines of digital I/O, 10/100 Base-T Ethernet, CRT and flat-panel video and PC/104 and PC/104-Plus expansion. A CompactFlash socket is available for bootable and removable memory, up to 2GB. Companion XE-800 OS Embedder kits, which include hardware and software for instant operation, are available for Linux 2.6 and QNX.

Octagon Systems, 6510 West 91st Avenue, Westminster, Colorado 80031, 303-430-1500, www.octagonsystems.com.

PathScale Compiler EKO Compiler Suite

PathScale announced the availability of the EKO Compiler Suite for AMD Opteron and Athlon 64 systems. The EKO Suite offers C, C++ and Fortran 9X compilers and beta support for 32-bit x86 compilation. The PathScale compilers

provide binary compatibility, with the ability to mix and match the linking of GNU GCC and PathScale compiled libraries and objects. The front ends are source-compatible with the GNU compiler suite for C/C++. The Fortran 95 compiler provides support for the most common Cray/SGI extensions, and in-line AMD64 assembly code also can be issued. The PathScale Compiler is available in installable Linux RPM format and is tested on SuSE, Red Hat and Fedora. The compilers can be purchased as subscriptions to the full EKO suite or to separate languages.

PathScale, Inc., 477 North Mathilda Avenue, Sunnyvale, California 94085, 408-746-9100, www.pathscale.com.

gumstix Boards and Computers

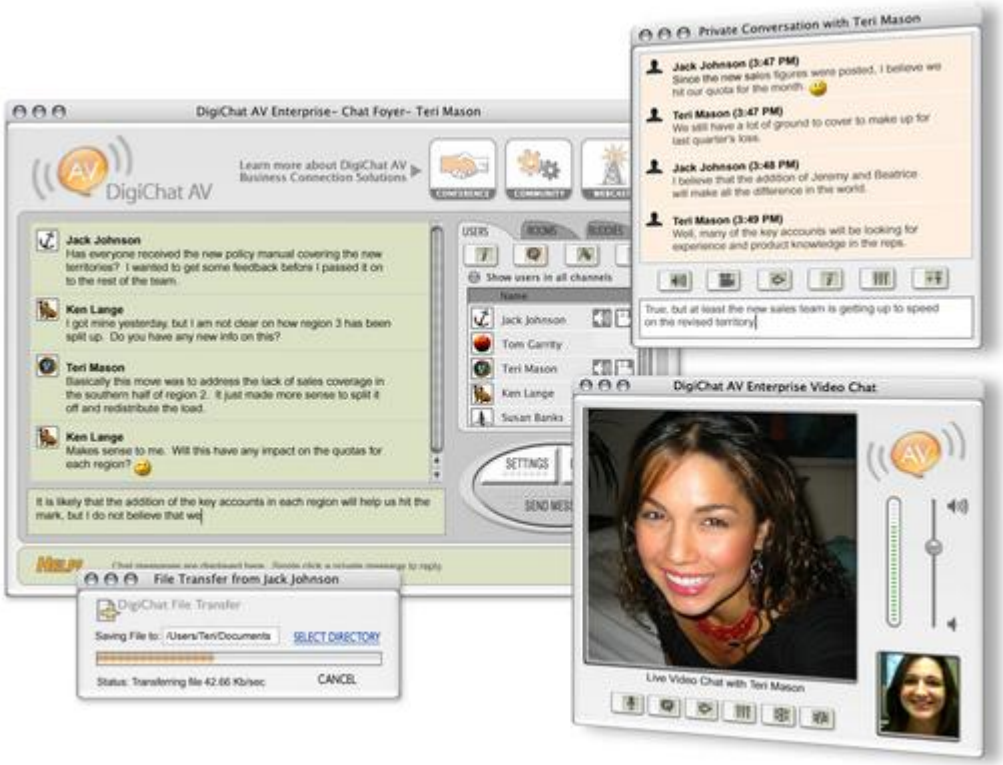
gumstix, Inc., introduced a new line of tiny Linux single-board processors (SBCs) and peripherals. Based on Intel's PXA255 processor with XScale technology, gumstix tiny boards measure 20mm × 80mm × 8mm. The line includes two gumxstix boards and two waysmall computers. The gumstix 200x and 400x boards feature 200MHz and 400MHz Intel PXA255, respectively; both offer 64MB of SDRAM, 4MB of Flash, an OS, an MMC.SDT slot and multiple I/Os. The waysmall 200x offers a gumstix 200x in a gumstix box, and the waysmall 400x offers a gumstix 400x. Both gumstix boxes feature two mini-DIN8 serial ports, one USB mini-B client port, a case and a power supply. The boards are stackable and draw less than 250mA at 400MHz. A GCC toolchain offers access to open-source software for porting. The boards ship with 4MB of Flash, containing u-boot-1.0.0, kernel 2.6.4 and a root filesystem. The computers include a BusyBox implementation with a Web server, a complete Linux kernel and a cross-compiler.

gumstix, Inc., www.gumstix.com.

DigiChat AV Enterprise 5.0

DigiChat AV Enterprise 5.0 is Java software that enables Web-based chatting, on-line collaboration, e-learning and moderated Webcasts. Features of version 5.0 include voice chat (VoIP), video chat (P2P), Web-based instant messaging support, a GUI with skinnable interfaces, a high-performance text messaging engine, HTTP tunneling support, scriptable BOTs support and scriptable command-line and Java APIs. The new client-side plugin architecture allows users to extend and create new programs within DigiChat. Version 5.0 also offers an integrated IM application that can be installed locally. Users can share text documents, PDFs, images, sounds and video directly through DigiChat. DigiChat supports UNIX/Linux, Windows 9x/NT/2000/XP, Mac OS/OS X and Solaris.

Digi-Net Technologies, Inc., 1034 Northwest 57th Street, Gainesville, Florida 32605, 877-404-2428, www.digi-net.com.



Visual SlickEdit 9

Version 9 of Visual SlickEdit, a development tool with a high-level code editor, offers ten C++ refactorings to enable developers to improve the structure of source code for better performance. Other new features in version 9 include a Java GUI builder, full-screen editing and dual-monitor support, a backup history that stores changes locally, CodeWright emulation and a new HTML Help and tutorial assistant. To simplify builds, Visual SlickEdit offers a C/C++ auto-build system as well as support for Ant. Visual SlickEdit includes integrated C/C++ and Java debuggers. The advanced code editor features Context Tagging, which offers language-specific coding assistance for a multitude of languages. The DIFFzilla differencing system, which provides side-by-side file and directory difference editing, works with three-way merge to support version control.

SlickEdit, Inc., 3000 Aerial Center Parkway, Suite 120, Morrisville, North Carolina 27560, 800-934-3348, www.slickedit.com.

Archive Index Issue Table of Contents

Advanced search