LINUX JOURNAL

Advanced search

Linux Journal Issue #49/May 1998

Archive Index

The Perl Debugger

Jeremy Impson

Issue #49, May 1998

The Perl debugger, a part of the core Perl distribution, is a useful tool to master, allowing close interactive examination of executing Perl code.

This article is a tutorial about the Perl5 source debugger and assumes that the reader has written at least one or more simple Perl programs. It is best read in front of a computer, following along with a copy of the code, available at *Linux Journal*'s FTP site (see Resources). The version of Perl that I use is perl5.004_1, which comes with the Perl debugger level 1. I've noticed some subtle differences between this and earlier versions of the debugger. If something discussed here doesn't work for you, consider upgrading.

The Perl programming language is being used increasingly on the World Wide Web as the back end to Common Gateway Interface (CGI) forms and interactive web pages, as well as for automated scripts for maintaining web sites and Unix servers in general. As a result, more and more users are beginning to learn Perl.

Conceptually, a debugger is a tool which allows the programmer a greater degree of control over the execution of the program without having to physically insert code that provides this control. A debugger allows the programmer to step through the program code, line by line if necessary. It allows peeks into the contents of the variables of the program, as well as into the stack, which is basically the list of functions (known as subroutines in Perl parlance) that have been called in order to get from the the *main* part of the program to the current point of execution.

Examples of Debuggers

There are many different debuggers. Some, such as **dbx** or **gdb**, are separate programs that can be used to debug programs written in languages such as C, C++, Modula-2 or FORTRAN. (**gdb**, for instance, can handle C, C++ and

Modula-2, according to its man page, which on my system dates from 1991, so by now it may cover FORTRAN.) Programming environments from Borland, Microsoft and others may have debugging capabilities built into their windowing environment.

Invoking the Debugger with Unix

Invoking the Perl debugger is as easy as invoking Perl itself. All one needs to do is provide the **-d** option when invoking the Perl interpreter, like this:

```
perl -d perlscript.pl
```

Invoking the Debugger with Windows

Perl has also been ported to Win32 systems and can be invoked similarly. If your system supports the **#!** syntax for scripts, you can have this as the first line of your Perl script (assuming you keep the Perl interpreter in /usr/bin):

```
#!/usr/bin/perl -d
```

This option isn't supported under Win32 systems (that I know of), but there are ways to simulate it. See the appropriate documentation.

Invoking the Debugger with Emacs

The Perl debugger can also run under Emacs, creating an integrated programming environment that is similar to products from Borland or others.

How the Debugger Works

Generally, when using a program written in Perl, you are invoking the program with the Perl interpreter. A Perl compiler is on the horizon, but will not be directly covered by this article. (The most logical way to debug code intended for the Perl compiler is to use the standard debugger until the code is “bug free”, then compile it.)

Under normal conditions, the Perl interpreter will read in the Perl script and will do a certain amount of compilation, turning your Perl code into some highly-optimized instructions, which are then interpreted. When using the debugger, extra Perl code is inserted into your code before it is handed off to the interpreter. Also, a library file, called in current releases perl5db.pl, is *required* in your Perl script. This final script is interpreted, resulting in the program running in the debugging environment.

The Warning Flag

When programming in Perl, you should probably always use the warning flag. Use this just as you would use the debug flag, as follows:

```
perl -w
```

When you are getting strange results from your program, you should **definitely** use the warning flag. The warning flag causes Perl to issue warnings regarding your code. These warnings are about things which are not fatal, but may cause problems. You can view the warnings as critiques of your coding style. Common warnings are those indicating that a certain variable has been used only once (perhaps a typo), or that a used package can't be found (maybe the package is not available on your system or is installed incorrectly).

Perl doesn't make you specify function prototypes and allows you to create variables at any point, so you don't have the advantages of type checking, although, with Perl 5 you can optionally have type checking for subroutines.

Commands (also see the man page)

This tutorial covers the debugger commands that I've found the most useful. The **perldebug** man page has a complete list of commands.

The most important command that can be entered into the debugger is **h**, which prints out a help screen. This tends to scroll off the screen, so type **h h** to see the help screen better formatted to fit your screen. Or, you can type **|h**, which will pipe the output of the command **h** into a pager, such as **more** or **less**. You can define what pager to use by setting the PAGER environmental variable to whatever pager you prefer. I prefer using less. (You can actually do this from within the debugger by typing:

```
$ENV{'PAGER'} = "/usr/bin/less"
```

at the debugger prompt.) This piping mechanism works with more than just the help command, so if you ever do something and the result moves off your screen, try prepending it with a pipe. You can get help with individual commands by typing **h command**.

Simple Example

Now let's look at actual examples of using the debugger. We'll start with the simple snippet of Perl code in Listing 1, called p1.pl. Notice that we are executing the code with the **-d** option to **perl**, invoking the debugger. Upon invoking the script under the debugger, we'll see the following:

```
Loading DB routines from perl5db.pl version 1
Emacs support available.
Enter h or h h for help.
main::(./p1.pl:3): if(0) {
 DB<1>
```

The debugger has suspended the normal execution of p1.pl, and is waiting for a command. Notice that we are given some information concerning where we are in the text of the program. The string **main::(./p1.pl:3):** tells us that we are in the *main* part of the Perl code, that the program we are executing is **./p1.pl** and that we are at line three of the code. If we were in the middle of another Perl package, that package name would be listed here. We are also shown that line three is **if(0) {**. When we see code on a line, we have not yet executed it; rather it is this line in the code that is about to be executed. The next line, **DB<1>**, is a prompt at which to enter the first command to the debugger. If you enter a command and wish to repeat it, you can enter **! *comnum***, where ***comnum*** is the command number you wish to repeat.

Listing Code

We can see more of the surrounding script by typing **l** and pressing **enter**. Be careful not to put white space before this or any other commands. Doing so tells the debugger that what follows is not a command. Instead, the debugger will try to execute the code as normal Perl code and will evaluate it in the current context of the program being debugged. The debugger will do the same thing for any input it doesn't recognize as a debugger command. Using the character ; (semicolon) to end the command is optional.

Entering **l** (letter l for list) causes the following lines to appear on the screen:

```
3==> if(0) {
4: print "Can't get here!\n";
5 }
6
7: while ($i < 10) {
8: $i++;
9 }
10
11: if($i >= 9) {
12: print "Hello, world!\n";
 DB<1>
```

Notice the arrow, **==>**. This represents the current line of code. In this case, it is line 3 and is the first actual line of Perl code. Notice also that all the lines which actually have executable code on them are labeled with a : (colon) after the line number. This is important, because later on when we get into breakpoints and action points, we will only be able to set them at these lines.

Entering **l** again yields this output:

```
13 }
14
```

```
15: exit 0;
 DB<1>
```

The **l** without any arguments reveals the next window of Perl code. Subsequent usage reveals the next window and the next. There is an internal line pointer that gets incremented one window each time **l** is used. To back up a window, type **-** (hyphen) and press **enter**, then press **l** again.

There are also arguments to the **l** command, dealing with various ways of specifying what lines are printed based on their line numbers. We will use some of them as we need them. Similar to **l** is **w**, which prints out windows of program text. See the perldebug man page for details.

Stepping Through Lines of Code

There are two ways to execute the code. We know that the current line is 3 and is an **if** statement. The first method, **s**, is to step through the code, statement by statement. The other method is **n**, for *next*, which similarly steps through the code; however, in the case where the current statement is a subroutine call (as opposed to a built-in function or some sort of variable assignment), **n** will treat the subroutine as though it were a built-in function and will step over the subroutine, as if it is an atomic command. In contrast, **s** will enter the subroutine and step through every line of the subroutine. It will do the same for any subroutines encountered within the first one. This can be annoying when we know that the subroutine is working correctly—hence the **n** command. For this simple example, where we have no subroutines, **n** has the same effect as **s**. After entering **s** or **n**, we can simply press **enter**, and the debugger will reissue the last **s** or the last **n** command. This is useful to get through lines of code quickly. Pressing **s** displays the following:

```
main::(./p1.pl:7): while ($i < 10) {
```

Notice that we've skipped from line 3 to line 7. Enter **l 3+4**. This shows us four lines from line 3. We skipped to line 7 because the conditional in line 3, **if(0)**, is false. So the **then** part of the conditional is ignored, and the **else** portion is executed.

Listing Variables

Notice that there is a variable in the code, **$i**. We know that the body of the while loop will be executed until **$i** is greater than or equal to 10. (Enter **l 7+10** to see the body of the while loop.)

So what value does **$i** have now? Type **p $i**. The print command is **p**, and without an argument; it will print the contents of the magic Perl variable **$_**. Any valid Perl expression is a valid argument to **p**. Because anything that the debugger doesn't recognize as a debugger command is evaluated as Perl code, you could also type **print** instead of **p**. Don't worry about having redirected standard output to something other than your screen. The debugger will take care to ensure that you'll see some output. But, typing **p** is quicker than typing **print**, and as any good programmer knows, laziness is one of the “programmer's virtues”, the other two being hubris and impatience (Larry Wall, see Resources).

Typing **p $i** results in nothing. No, we didn't do anything wrong. **$i** hasn't been set to anything, so it gets the default value of nothing. Type **s** (or just press **enter**). Try **p $i** again. It should print the number 1. Press **enter** again and type **p $i** again. Now, we could continue this, but we know that we will keep spinning in this *while* loop until the conditional returns false, which won't happen until **$i** is no longer less than 10. And, as I said before, impatience is another programmer's virtue, so we'll rush things along a bit. Enter **$i = 8**, then press **enter** again. Do it one more time, and we've broken free of the loop.

The last conditional checks that **$i** is at least equal to 9. Because it now is, the **then** portion of the **if** statement will not get executed. Note that we could have set **$i** back to 2 before we execute the final **if** statement. The result would have been an execution that under normal conditions (i.e., without using the debugger) could never have occurred (assuming the computer is working properly, and no bits in memory get fiddled).

As any good first program should, our first debug program prints **Hello, World!** to the screen. Notice that even under the debugger, this happens as it should. Pressing **enter** one more time terminates the program.

More Complex Example

The code in Listing 2 is a more complex piece of code with a bug in it. It should print out every regular file in the current directory and all subdirectories, recursively. Right now, it only prints the files in the current directory and doesn't seem to delve into further subdirectories.

Execute this program in a directory with a few subdirectories and place files and further subdirectories in these subdirectories to create a small but diverse hierarchy.

The output of this code (once the bug gets fixed) from the directories I ran it in, looked like this:

```
./file1
./dir1.0/file1
./dir1.0/file2
./dir1.0/file3
./dir1.0/dir1.1/file1
```

```
./dir1.0/dir1.1/file2
./dir1.0/dir1.1/file3
./dir2.0/file1
./dir2.0/file2
./dir2.0/file3
./dir2.0/dir2.1/file1
./dir2.0/dir2.1/file2
./dir3.0/file1
```

Subroutines

There is one more variation of the list code command, **l**. It is the ability to list the code of a subroutine, by typing **l *sub***, where ***sub*** is the subroutine name.

Running the code in Listing 2 returns:

```
Loading DB routines from perl5db.pl version 1
Emacs support available.
Enter h or h h for help.
main::(./p2.pl:3): require 5.001;
 DB<1>
```

Entering **l searchdir** allows us to see the text of **searchdir**, which is the meat of this program.

```
22 sub searchdir { # takes directory as argument
23: my($dir) = @_;
24: my(@files, @subdirs);
25
26: opendir(DIR,$dir) or die "Can't open \"
27:     $dir\" for reading: $!\n";
28
29: while(defined($_ = readdir(DIR))) {
30: /^\./ and next; # if file begins with '.', skip
31
32 ### SUBTLE HINT ###
```

As you can see, I left a subtle hint. The **bug** is that I deleted an important line at this point.

Setting Breakpoints

If we were to step through every line of code in a subroutine that is supposed to be recursive, it would take all day. As I mentioned before, the code as in Listing 2 seems only to list the files in the current directory, and it ignores the files in any subdirectories. Since the code only prints the files in the current, initial directory, maybe the recursive calls aren't working. Invoke the Listing 2 code under the debugger.

Now, set a breakpoint. A breakpoint is a way to tell the debugger that we want normal execution of the program until it gets to a specific point in the code. To specify where the debugger should stop, we insert a breakpoint. In the Perl debugger, there there are two basic ways to insert a breakpoint. The first is by line number, with the syntax **b linenum**. If **linenum** is omitted, the breakpoint is inserted at the next line about to be executed. However, we can also specify breakpoints by subroutine, by typing **b** ***sub***, where ***sub*** is the subroutine name. Both forms of breakpointing take an optional second argument, a Perl conditional. If when the flow of execution reached the breakpoint the conditional evaluates to true, the debugger will stop at the breakpoint; otherwise, it will continue. This gives greater control of execution.

For now we'll set a break at the **searchdir** subroutine with **b searchdir**. Once the breakpoint is set, we'll just execute until we hit the subroutine. To do this, enter **c** (for continue).

Adding Actions

Looking at the code in Listing 2, we can see that the first call to **searchdir** comes in the main code. This seems to works fine, or else nothing would be printed out. Press **c** again to continue to the next invocation of **searchdir**, which occurs in the **searchdir** routine.

We wish to know what is in the **$dir** variable, which represents the directory that will be searched for files and subdirectories. Specifically, we want to know the contents of this variable each time we cycle through the code. We can do this by setting an action. By looking at the program listing, we see that by line 25, the variable **$dir** has been assigned. So, set an action at line 25 in this way:

```
a 25 print "dir is $dir\n"
```

Now, whenever line 25 comes around, the **print** command will be executed. Note that for the **a** command, the line number is optional and defaults to the next line to be executed.

Pressing **c** will execute the code until we come across a breakpoint, executing action points that are set along the way. In our example, pressing **c** continuously will yield the following:

```
main::(../p2.pl:3): require 5.001;
 DB<1> b searchdir
 DB<2> a 25 print "dir is $dir\n"
 DB<3> c
main::searchdir(../p2.pl:23): my($dir) = @_;
 DB<3> c
dir is .
main::searchdir(../p2.pl:23): my($dir) = @_;
 DB<3> c
dir is dir1.0
main::searchdir(../p2.pl:23): my($dir) = @_;
 DB<3> c
dir is dir2.0
main::searchdir(../p2.pl:23): my($dir) = @_;
 DB<3> c
dir is dir3.0
file1
file1
file1
file1
DB::fake::(/usr/lib/perl5/perl5db.pl:2043):
2043: "Debugged program terminated. Use `q' to quit or `R' to
restart.";
 DB<3>
```

Note that older versions of the debugger don't output the last line as listed here, but instead exit the debugger. This newer version is nice because when the program has finished it still lets you have control so that you can restart the program.

It still seems that we aren't getting into any subdirectories. Enter **D** and **A** to clear all breakpoints and actions, respectively, and enter **R** to restart. Or, in older debugger versions, simply restart the program to begin again.

We now know that the **searchdir** subroutine isn't being called for any subdirectories except the first level ones. Looking back at the text of the program, notice in lines 44 through 46 that the only time the searchdir subroutine is called recursively is when there is something in the **@subdirs** list. Put an action at line 42 that will print the **$dir** and **@subdirs** variables by entering:

```
  a 42 print "in $dir is @subdirs \n"
```

Now, put a breakpoint at line 12 to prevent the program from outputting to our screen (**b 12**), then enter **c**. This will tell us all the subdirectories that our program thinks are in the directory.

```
  main::(../p2.pl:3): require 5.001;
   DB<1> a 42 print "in $dir is @subdirs \n"
   DB<2> b 12
   DB<3> c
  in . is dir1.0 dir2.0 dir3.0
  in dir1.0 is
  in dir2.0 is
  in dir3.0 is
  main::(../p2.pl:12): foreach (@files) {
   DB<3>
```

This program sees that there are directories in “.”, but not in any of the subdirectories within “.”. Since we are printing out the value of **@subdirs** at line 42, we know that **@subdirs** has no elements in it. (Notice that when listing line 42, there is the letter “a” after the line number and a colon. This tells us that there is an action point here.) So, nothing is being assigned to **@subdirs** in line 37, but *should* be if the current (as held in **$_**) file is a directory. If it is, it should be pushed into the **@subdirs** list. This is not happening.

One error I've committed (intentionally, of course) is on line 38. There is no catch-all “else” statement. I should probably put an error statement here. Instead of doing this, let's put in another action point. Reinitialize the program so that all points are cleared and enter the following:

```
a 34 if( ! -f $_ and ! -d $_ ) { print "in $dir: $_ is
weird!\n" }
b 12"
c
```

which reveals:

```
main::(../p2.pl:3): require 5.001;
 DB<1> a 34 if( ! -f $_ and ! -d $_ ) { print "in $dir:
$_ is weird!\n" }
 DB<2> b 12
 DB<3> c
in dir1.0: dir1.1 is weird!
in dir1.0: dir2.1 is weird!
in dir1.0: file2 is weird!
in dir1.0: file3 is weird!
in dir2.0: dir2.1 is weird!
in dir2.0: dir1.1 is weird!
in dir2.0: file2 is weird!
in dir2.0: file3 is weird!
main::(../p2.pl:12): foreach (@files) {
 DB<3>
```

While the program can read (through the **readdir** call on line 29) that dir1.1 is a file of some type in dir1.0, the file test (the **-f** construct) on dir1.1 says that it is not.

It would be nice to halt the execution at a point (line 34) where we have a problem. We can use the conditional breakpoint that I mentioned earlier to do this. Reinitialize or restart the debugger, and enter:

```
b 34 ( ! -f $_ and ! -d $_ )
c
p
p $dir
```

You'll get output that looks like this:

```
main::(../p2.pl:3): require 5.001;
 DB<1> b 34 ( ! -f $_ and ! -d $_ )
 DB<2> c
main::searchdir(../p2.pl:34): if( -f $_) { # if its a file...
 DB<2> p
dir1.1
 DB<2> p $dir
dir1.0
 DB<3>
```

The first line sets the breakpoint, the next **c** executes the program until the break point stops it. The **p** prints the contents of the variable **$_** and the last command, **p $dir** prints out **$dir**. So, dir1.1 is a file in dir1.0, but the file tests (**-d** and **-f**) don't admit that it exists, and therefore dir1.1 is not being inserted into **@subdirs** (if it's a directory) or into **@files** (if it's a file).

Now that we are back at a prompt, we could inspect all sorts of variables, subroutines or any other Perl construct. To save you from banging your heads against your monitors, and thus saving both your heads and your monitors, I'll tell you what is wrong.

All programs have something known as the current working directory (CWD). By default, the CWD is the directory where the program starts. Any and all file accesses (such as file tests or file and directory openings) are made in reference from the CWD. At no time does our program change its CWD. But the values returned by the **readdir** call on line 29 are simply file names relative to the directory that **readdir** is reading (which is in **$dir**). So, when we do the **readdir**, **$_** gets assigned a string representing a file (or directory) within the directory in **$dir** (which is why it's called a subdirectory). But when running the **-f** and **-d** file tests, they look for **$_** in the context of the CWD. But it isn't in the CWD, it's in the directory represented by **$dir**. The moral of the story is that we should be working with **$dir/$_**, not just **$_**. So the string

```
###SUBTLE HINT###
```

should be replaced by

```
$_ = "$dir/$_"; # make all path names absolute
```

That sums it up. Our problem was we were dealing with relative paths, not absolute (from the CWD) paths.

Putting it back into our example, we need to check **dir1.0/dir1.1**, not **dir1.1**. To check to make sure that this is what we want, we can put in another action point. Try typing:

```
a 34 $_ = "$dir/$_"
c
```

In effect this temporarily places the corrective measure into our code. Action points are the first item on the line to be evaluated. You should now see the proper results of the execution of the program:

```
DB<1> a 34 $_ = "$dir/$_"
DB<2> c
./file1
./dir1.0/file1
./dir1.0/file2
./dir1.0/file3
./dir1.0/dir1.1/file1
./dir1.0/dir1.1/file2
./dir1.0/dir1.1/file3
./dir2.0/file1
./dir2.0/file2
./dir2.0/file3
./dir2.0/dir2.1/file1
./dir2.0/dir2.1/file2
./dir3.0/file1
DB::fake::(/usr/lib/perl5/perl5db.pl:2043):
2043: "Debugged program terminated. Use `q' to quit or `R' to
restart.";
 DB<2>
```

Stack Traces

Now that we've got the recursive call debugged, let's play with the calling stack a bit. Giving the command **T** will display the current calling stack. The calling stack is a list of the subroutines which have been called between the current point in execution and the beginning of execution. In other words, if the main portion of the code executes subroutine “a”, which in turn executes subroutine “b”, which calls “c”, then pressing “T” while in the middle of subroutine “c” outputs a list going from “c” all the way back to “main”.

Start up the program and enter the following commands (omit the second one if you have fixed the bug we discovered in the last section):

```
b 34 ( $_ =~ /file2$/)
a 34 $_ = "$dir/$_"
c
```

These commands set a breakpoint that will only stop execution if the value of the variable **$_** ends with the string **file2**. Effectively, this code will halt execution at arbitrary points in the program. Press **T** and you'll get this:

```
@ = main::searchdir('./dir1.0/file2') called from file '../p2.pl' line
45
@ = main::searchdir(.) called from file '../p2.pl' line 10
```

Enter **c**, then **T** again:

```
@ = main::searchdir('./dir1.0/dir1.1/file2') called from file
`../p2.pl' line 45
@ = main::searchdir(undef) called from file '../p2.pl' line 45
@ = main::searchdir(.) called from file '../p2.pl' line 10
```

Do it once more:

```
@ = main::searchdir('./dir2.0/file2') called from file '../p2.pl' line
45
@ = main::searchdir(.) called from file '../p2.pl' line 10
```

You can go on, if you so desire, but I think we have enough data from the arbitrary stack dumps we've taken.

We see here which subroutines were called, the debugger's best guess of which arguments were passed to the subroutine and which line of which file the subroutine was called from. Since the lines begin with **@ =** , we know that **searchdir** will return a list. If it were going to return a scalar value, we'd see **$ =**. For hashes (also known as associative arrays), we would see **% =**.

I say “best guess of what arguments were passed” because in Perl, the arguments to subroutines are placed into the @_ magic list. However, manipulating @_ (or **$_**) in the body of the subroutine is allowed and even encouraged. When a **T** is entered, the stack trace is printed out, and the current value of @_ is printed as the arguments to the subroutine. So when @_ is changed, the trace doesn't reflect what was actually passed as arguments to the subroutine.

Warnings

Well, by now you must be thinking, “Gosh, this Perl debugger is so keen that with it I can end world hunger, learn to play the piano and increase my productivity by 300%!” Well, this is the right attitude. You are now displaying the third programmer's virtue, hubris. However, there are some warnings.

Race Conditions

Race conditions are the scourge of the programmer. Race conditions are bugs that occur only under certain circumstances. These circumstances usually involve the time at which certain events correlate with other events. Using the debugger to debug these situations is not always possible, because the act of using the debugger may change the timing of the events in the program. This can cause a symptom to occur without the debugger, but while using the debugger, the symptom may disappear. The bug isn't gone, it just isn't being “tickled”.

There really isn't any stock method to get rid of race conditions. Usually, an intense analysis of the algorithms is necessary. Finite-state diagrams may also be useful, if you have the patience for it.

Process Management, IPC

When writing code that involves more than one process (for example, if your code uses a “fork” system call or its equivalent), using the debugger becomes very difficult. This is because when the fork occurs, you are left with two (or more) processes, all running under the debugger. But since the debugger is interactive, you have to interact with every process. The result is that you have to individually deal with each process, controlling each execution. All the processes will want to read debug commands from the controlling terminal, but only one at a time will be able to do so. The other(s) will block, waiting for the first to complete. When it does, another process will complete. Incidentally, we can't know for sure which process will be first. This is an example of the above mentioned race condition.

Perl Code Must Be Compilable

The final concern with using the debugger is compilation. Because the debugger is actually just debugging code inserted into your script, it is necessary that your script be compilable. That is, there should be no syntax errors.

Summary

Mastering the Perl 5 debugger is almost as useful as mastering Perl 5 itself. It allows you to take part in the actual execution of your program, to examine and experiment. It allows you to kill the bugs.

Jeremy Impson is a Senior Computer Science student at Syracuse University, in Syracuse, NY, studying Operating Systems. He spent the past summer working for IBM Global Services in Poughkeepsie, NY. He's been playing with Linux since Spring 1995, and has been hacking Perl just as long. Outside of computing and sleeping, he spends time studying history and cooking up strange recipes. You can reach him at jdimpson@acm.org.

Archive Index Issue Table of Contents

Advanced search

Building Reusable Java Widgets

R.J. Celestino

Issue #49, May 1998

An introduction to writing pluggable do-it-yourself widgets for the Java programmer.

There is no getting around it, if you are programming with Java, eventually you will need a specialized widget. I know, Sun Microsystems provides the Abstract Windowing Toolkit (AWT) with every download of Java; however, the AWT's small set of widgets and fairly low-level graphics will eventually leave you wanting. When it's time to wow the boss, the customer or the dog, it's time to roll up your sleeves and do it yourself.

This article examines some good techniques for extending the AWT. You will not simply learn *how* to create interesting widgets. You will also (and perhaps more importantly) learn how to make them *reusable* by employing good design practices and adhering to the AWT's architecture. To do this I will introduce some of the important concepts of the AWT's event model, hierarchy and graphics. [Perhaps the most significant change from version 1.0x to 1.1x was the shift from an inheritance-based event model to a more powerful delegation-based model.] This article will deal solely with Java version 1.1x and above, including the new event model.

I have written this article with the expectation that you have some basic experience with Java. You will need to know how to compile and run Java code to try out the examples. It would be a bonus if you have created the obligatory “Hello World” applet and are familiar with layout managers.

There is nothing Linux-specific in this article. One of Java's primary strengths is that it will run on any platform that has the Java virtual machine ported to it. Not surprisingly, Linux is one. In fact, chances are good that your version of Linux has Java compiled into the kernel. I have run these examples on various versions of Linux and Solaris 2.5. If you wish to experiment with building widgets on Linux, you will need to get a Linux port of the Java Development Kit

(JDK) from Blackdown or sunsite. Of course, you will be able to run the example applets in any Java-enabled browser that runs on Linux—that would be Netscape.

It Ain't Ugly, It's My GUI

If you write your GUI relying solely on the AWT, chances are pretty good that it *will* be ugly. While the AWT provides an excellent framework for GUI development, it was never meant to be the last word in GUI toolkits. The AWT provides a set of “least common denominator” widgets, so if you are interested in simple push buttons, sliders or text fields, the AWT has you covered. Unfortunately, modern user interfaces often call for GUI controls beyond these old standbys.

When your design requirements or creative wishes exceed the capabilities of these standard widgets, you need to create your own. When the time comes that you are ready to create your own widget, you will have choices. You *could* create a very uncooperative widget which is tightly coupled with your application. Or, you could choose to leverage the power of object-oriented programming, and the existing architecture of the AWT. It should be no surprise that I advocate the latter. If you agree, you will create open, reusable widgets that can be used by anyone in the Java community, regardless of his choice of operating system.

All the Other Widgets are Doing It

Linux users are a generally individualistic lot. Individuality has its place, but when it comes to the architecture of your widget I advocate fitting in with the crowd. When I go to the trouble of creating a widget I want to make sure that it can be used over and over painlessly. I achieve this by adhering to the design of the AWT. Although the AWT is not the most visually appealing widget toolkit, it does provide an excellent framework for widget creation and interaction. By working within the bounds of this framework we adhere to the basic architecture of the AWT and get reusable widgets.

When you design your widgets to cooperate with the existing architecture, you ensure that they are robust, maintainable and easily reusable by any Java developer (including yourself). Your widgets will be “pluggable”. Any Java user will be able to simply plug in the widget and use it like any other standard GUI componet provided in the AWT.

Foundations of a Reusable Widget

There are several important aspects to keep in mind when designing and creating widgets. In this section I will discuss some of these and evolve a template for widget creation.

Superclass

In order to cooperate with the AWT, your widget must have *Component* as its ancestor. I don't mean to imply that you must subclass Component directly; here in fact, you can often inherit some useful behavior by subclassng further down the hierarchy (see Figure 1 for a portion of the Component hierarchy).

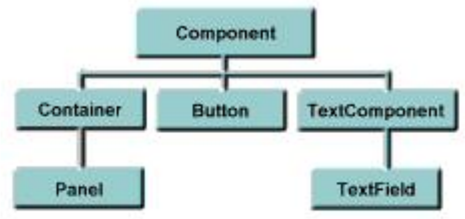


Figure 1. A Portion of the AWT's Component Hierarchy

Your widget might be drawn on the fly or created from images stored on a local or remote machine (GIFs for example). The Canvas class is a good choice for this type of component. The Canvas class provides a blank area for drawing. It will take some experience to be able to choose an appropriate superclass for your widget. It is wise to take some time to study the Component hierarchy of the AWT before choosing a superclass.

Events and Delegation

If your widget generates events, you have to stay within the bounds of Java's event model. In the “early days”, Java used a cumbersome inheritance-based event model. With the introduction of Java 1.1, the AWT sports a completely modern, delegation-based event model. The concept is simple. When a user interacts with a widget, it generates events. Objects can register an interest in all or some of these events. These interested objects, known as *listeners*, receive the events and take appropriate action. This process is known as delegation. Listeners are *delegated* by the widget to handle the events they generate.

When you create event classes, make sure they contain appropriate information and are generated in response to appropriate user actions. It can get confusing when you start to generate your own events. First, you create the *event classes*. Next, you create a *listener interface*. This interface defines methods of the listeners that will be called when an event occurs. Finally, you create an *event multicaster*. Event multicasters have the job of broadcasting events to many listeners. Sounds complicated, but hopefully it will become clear after an example or two. Figure 2 is a diagram of the multicasting process.

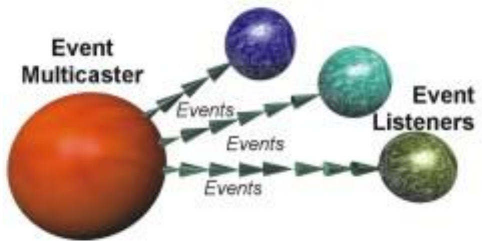


Figure 2. Event Multicaster

Drawing

The appearance of your widget will not strictly affect its ability to be reused. However, a widget's appearance should be in visual harmony with the rest of your GUI. Here are some things to think about, which, while not hard and fast rules, are important.

Modern widgets produce a 3D appearance by shading. Shading is created by an imaginary light source that is positioned, by convention, at the upper left of the widget. When a button is in the raised state, its upper and left borders are brighter than the face of the button (see Figure 3). Its lower and right borders are darker. This shading makes it appear to be raised. Swap the light and dark areas, and the button will appear depressed (see Figure 4).

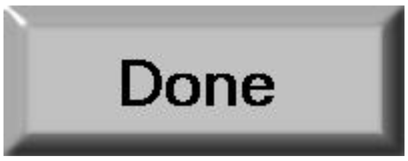


Figure 3. A Button in its Raised State

If your widget is drawn or has drawn areas on it, keep in mind the 3D effects are what make it as attractive as possible. And remember, drawing occurs in the **paint** method of your class. The AWT calls the paint method any time your widget needs to be drawn.

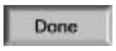


Figure 4. A Button in its Pressed State

Widgets to Call Your Own

There are generally two methods you can use to create reusable widgets: composition and specialization. Before continuing, let's take a moment to discuss each method.

Composition Widgets that are created by composition are sometimes called super-widgets or composite widgets. This type of widget is simply a collection of other widgets that work together to accomplish a specialized task. You should create a composite widget whenever you have a recurring task which requires a number of sub-components working together. Some examples include an order form, file dialog or a color chooser. When you create a composite widget, it is important to hide the events of all its sub-components and generate events at a higher level appropriate to the *semantics* of the composite widget itself. You will see how this works in the e-mail entry widget and the window-bar widget.

Specialization

Sometimes a widget is required that is slightly different from a standard AWT widget. Perhaps you need to add some new behavior or look. It is in these situations that you should consider creating a widget by specialization. When you create a widget by specialization, you create a subclass and add or override existing behavior. The power of inheritance gives you all of the behavior of the superclass, so all you need to do is write the new code. An example we will discuss is the VerticalSeparator, a subclass of Canvas. It overrides the **paint** method to achieve its own special look. The vertical-separator is a good example of this. The collapsing pane is a more subtle example of how to use specialization.

Widget Examples

For the sake of clarity in the examples, I have kept the graphics to a minimum and focused on the “nuts and bolts” of creating a reusable widget.

A note about style: I precede all of my instance variables with an underscore. Variables in upper case are class variables (static variables), and variables in lower case are generally temporary variables that have either been passed into the method or defined within the current block. Class and interface names always start with an upper case letter. Event classes always use “Event” as a suffix, and listener interfaces always have “Listener” for a suffix.

Example 1. Vertical Separator

The first widget is a simple vertical separator. A separator should be familiar to HTML users; the **<hr>** tag creates a horizontal rule. Separators are used to separate groups of components. The vertical separator is a very simple widget. It does not need to interact with any other widgets or objects. It is simply a visual component. After a little thought I came to the conclusion that this widget should be created by specialization, specifically by subclassing the Canvas class. Canvas is a good choice here since the only duty of the vertical separator is to render itself on the screen. Separators generally have an etched look to them, as if they were carved into the screen.

1. Rendering an Etched Line

To create the 3D effect of etching, draw two lines next to each other, one darker than the background, the other lighter. Here is a simple Java code fragment to create two vertical lines—one will appear etched, the other one raised. See Figure 5 for a picture of the two lines.

```
public void paint( Graphics g ) {
        // draw a raised line
        g.setColor( _light ) ;
        g.drawLine( 5, 10, 5, 40 ) ;
        g.setColor( _dark ) ;
        g.drawLine( 6, 10, 6, 40 ) ;
        // draw an etched line
        g.setColor( _dark ) ;
        g.drawLine( 25, 10, 25, 40) ;
        g.setColor( _light ) ;
        g.drawLine( 26, 10, 26, 40) ;
        }
```

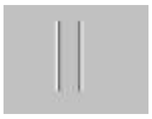


Figure 5. Etched and Raised Lines

Here is how I set the values of the two instance variables **_light** and **_dark** :

```
_light = getBackground().brighter().brighter() ;
_dark  = getBackground().darker().darker() ;
```

Setting the values *relative* to the background color rather than a hard-coded color makes the code more general. These lines will appear etched and raised regardless of the background color of the region in which they appear.

1.2. Sizing the Separator

The vertical separator should fill its allocated space vertically and center in its space horizontally. Within the **paint** method you can determine the space that has been allotted and calculate its dimensions. Here is how I did it:

```
size = size() ;
int length = size.height ;
int yPosition = ( size.width )/2 ;
g.setColor( dark ) ;
g.drawLine( 0, yPosition, length, yPosition ) ;
g.setColor( light ) ;
g.drawLine( 0, yPosition+1, length, yPosition+1 );
```

Now, there is one more critical method we need to override. Remember that I have chosen to subclass Canvas. The Canvas class provides a default size of 0x0. This means that if the widget is laid out using its default size, it will not show up. To achieve a meaningful default size, you need to override the **getPrefferedSize** and **getMinimumSize** methods. I have chosen a region of 4x8 pixels for both its preferred size and its minimum size. Why did I choose 4x8? The separator has an actual width of 2 pixels. Setting its preferred width to 4 gives it a 1 pixel buffer on each side. The preferred height of 8 pixels is somewhat arbitrary—any value greater than 0 is acceptable, just so it is visible. Remember if the separator is used properly, the layout manager will grow its height to an appropriate value regardless of the preferred height.

You can see the completed VerticalSeparator class in Listing 1. Remember, we built the vertical separator to fill the vertical space that is allocated, so be sure to place it appropriately. The east and west portions of the *border layout* are guaranteed to be vertical regions. If the separator is placed in either of those regions, it will grow to fill the vertical region. If you place it in the north or south regions, it will be sized to its preferred height of 8 pixels, which may not be what you want. I recommend you read up on layout managers and how they respond to a widget's need to be sized. Some completely disregard preferred sizes. You can see an applet that uses the vertical separator in Figure 6.

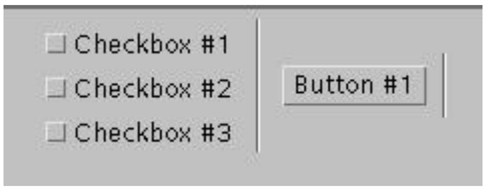


Figure 6. The VerticalSeparator in Action

In this example I have introduced some fundamental concepts of drawing and sizing. The goal of the vertical widget is simply to make your GUI look better. This is a noble goal, but in the next few examples we are going to look at some harder working widgets that really earn their keep. Before looking at those widgets, here are a couple of challenges.

1.3. Exercises for the Reader

There are a few interesting extensions you could make to this class, and I leave them to you:

- **Generic separator**: To create a horizontal separator you might want to create a HorizontalSeparator class. But why not think about creating a single class that does both depending on how it is placed? It would need to know if it were placed in a vertical or horizontal region and render itself appropriately.
- **Additional features**: The separator we created has a fixed width of 2 pixels and is etched. Change or extend your class to support a variable width and the option of etched, raised or flat.

Example 2. E-mail Entry Widget

The e-mail entry widget represents a simple entry form used to gather information from a user. It is a composite widget built with standard AWT components. This example widget is important in the way that it interacts with other classes. This class broadcasts customized events which can be captured by any interested object in the system. The ability to broadcast custom events to listeners of those events is critical in creating a reusable widget. The e-mail widget will have a **done** and **cancel** button. Remember, you do not want other objects to have direct access to these buttons or the events that they generate. Why? Suppose the e-mail widget changes down the line and no longer uses a button click to signify completion. Every object that uses the class will break and will have to be rewritten. If you hide these events, how can your widget notify interested parties of its changes in state? You must generate widget specific events and create listeners of these events. These events must make semantic sense to the action of the e-mail widget. We will see how this is done in a moment, but first let's create its visual appearance.

2.1. Layout

The visual layout of the e-mail entry form is created in its constructor. The widget consists of a text field, a label and two buttons.

```
Class EmailEntry extends Panel implements
        ActionListener {
public EmailEntry() {
        super() ;
        _doneButton = new Button( "Done" );
        _cancelButton = new Button( "Cancel" ) ;
        _emailField = new TextField( 40 ) ;
        // build a sub-panel for the buttons.
        Panel = new Panel() ;
        buttonPanel.add( _doneButton ) ;
        buttonPanel.add( _cancelButton ) ;
        // install the components in the widget
        this.setLayout( new BorderLayout() ) ;
        this.add( "West", new Label(
                "Enter your e-mail address"));
        this.add( "Center", _emailField) ;
        this.add( "South", buttonPanel ) ;
        // forward events to myself
        _doneButton.addActionListener( this ) ;
        _cancelButton.addActionListener( this )
        }
}
```

Notice that in the widget subclasses **Panel**, I chose panel as the superclass so that I can inherit its layout capabilities. Also notice the class implements the **ActionListener** interface. This means that the class is allowed to listen to action events (the events that are generated by buttons). Toward the end of the constructor the class registers itself (in the call to **addActionListener**) as a listener of both push buttons.

2.2. Create the Event Classes

You must determine the events that your widget will generate. I have chosen to create a single event class, the *EmailEntryEvent* that can represent either “user is done” or “user canceled” state. When you create your event class keep in mind that it must maintain sufficient information to act on the event. In this example the event must store the e-mail address that was entered. Listing 2 shows the EmailEntryEvent class. Notice that I have created two constructors. When the constructor is passed an e-mail string, an e-mail entry event of type **done** is created. If no information is passed to the constructor, an event of type **cancel** is created. The e-mail entry widget has the responsibility of invoking the proper constructor (a reasonable request of the widget).

2.3. Create a Listener Interface

When an event is broadcast to a listener, specific methods of the listener class are invoked. You might think of these as “callbacks”. The *listener interface* defines these methods. The interface ensures that any class intended to be a listener has the methods needed to handle the event. The e-mail entry listener interface is shown in Listing 3. Any class that wishes to be a listener of EmailEntryEvents is required to implement a **done** method and a **cancel** method.

2.4. Event Multicasters

Event multicasters handle asynchronous broadcasting of events to listeners. You will be relieved to know that you do not have to write your own multicaster from scratch. Instead you will need to create a subclass of the existing *AWTEventMulticaster* and write a few methods so that it can handle your new events. When you create your multicaster follow these steps:

- *Implement* the listener interface that you created in Listing 3.
- Create the **add** and **remove** methods using that same listener.
- Create the methods defined in the listener interface. These methods simply forward the messages to the appropriate listeners.

Take a look at Listing 4 to see how I created the multicaster for this example. Don't let the multicaster scare you. The code is very basic, cookie-cutter-style. You will be relying on the superclass to do all of the hard multicasting work. All that your multicaster must do is be aware of your new events and listeners.

2.5. Hooking it all up

We have now created all of the necessary components,, and what remains is to connect it up properly. The first order of business is to complete the

EmailWidget class. As we left it, it only had a constructor. Next you must give it the ability to add listeners. Here is the *addEmailEntryEventListener* method:

```
public void addEmailEntryListener(
        EmailEntryListener e ) {
_emailEntryListener = MyMulticaster.add(
        _emailEntryListener, e ) ;
}
```

Notice that the widget has an instance variable that maintains its listener. If you look closely, you will see that this variable is actually an instance of your multicaster class. Any widget can potentially have many listeners. Your multicaster will maintain the list of listeners and ensure that they get their appropriate events. Finally, you must handle the internal events (from the buttons) and generate your new event.

```
public void actionPerformed( ActionEvent e ) {
        EmailEntryEvent newEvent ;
        if ( _emailEntryListener == null ) return ;
        if ( e.getSource() == _doneButton ) {
                newEvent = new EmailEntryEvent(
                                getEmailAddress()) ;
        _emailEntryListener.done( newEvent );
    }
        else if ( e.getSource() == _cancelButton ) {
                newEvent = new EmailEntryEvent() ;
                _emailEntryListener.cancel( newEvent );
    }
```

In this code you generate the widget specific events. When the **done** button is pressed, a “done” event is created that stores the e-mail address. Otherwise a “cancel” event is generated. Notice also how the event is dispatched: the corresponding method is called on the multicaster (**_emailEntryListener** is an instance of **MyMulticaster**). The multicaster then forwards the method call to all of the registered listeners. The EmailEntry widget is now ready to be used by any Java program. Listing 5 shows a simple applet that uses the e-mail entry widget and intercepts the events. Take a look at the e-mail entry widget in Figure 7.

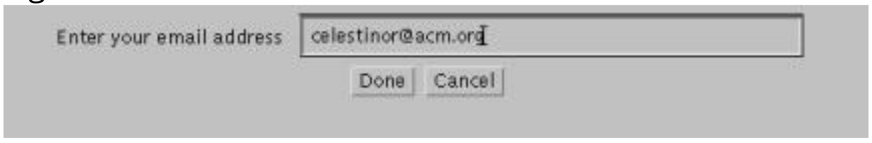


Figure 7. The EmailEntryWidget in Use as an Applet

One important note: the e-mail entry class generates a done event. How is this different from the action event generated by the **done** button? The difference is subtle, but important. If you rely upon detecting the events of a specific button, other classes need to know the details of the inner workings of your class. If this changes, every class that uses it has to change as well.

2.6. Exercises for the Reader

Here are a few ideas for expanding the EmailEntry widget that you might enjoy trying:

- **Collect more information**: Expand the widget by collecting more information from the user. Add checkboxes, additional text fields and so on. Remember that your *event class* has to maintain this information and pass it on to the listeners.
- **Add error checking**: Check for things like the e-mail address being complete and reasonable. Be sure that you are broadcasting the event only if the form is completed correctly.

Example 3. WindowBar

The window bar is a component needed for the collapsing pane example below. You click on the window bar and the pane collapses. Click on it again and it opens up. To keep it simple I will use a button for the bar, rather than creating a fancy visual bar. The bar must broadcast events to signal that the pane should collapse or restore.

3.1. Layout

The layout of this widget is intentionally trivial, to keep the example simple.

```
public WindowBar() {
        super() ;
        _closer = new Button( "Collapse" ) ;
        _closer.addActionListener( this ) ;
        add( _closer ) ;
}
```

3.2. Create Event Classes

For this widget we need an event with states of “collapse” and “restore”. This will be very similar to our e-mail entry widget. Here is the PaneSwitchEvent:

```
class PaneSwitchEvent extend AWTEvent
{
  public static final int COLLAPSE = 1 ;
  public static final int
  RESTORE =2 ; private int _type ;
  public PaneSwitchEvent( Object source, int t )
  {
        super( source , 0 ) ; _type = t ;
  }
  public boolean isRestore()
  {
        return _type == RESTORE ;
  }
}
```

The pane switch event needs only to maintain the type of event that it represents.

3.3. Create a Listener Interface

Let's continue down a familiar road and look at the PaneSwitchEventListener. Here it is:

```
interface PaneSwitchListener extends EventListener
{
  public void restore( PaneSwitchEvent e ) ;
  public void collapse( PaneSwitchEvent e ) ;
}
```

Your listener defines the two methods invoked when the pane switch event occurs. In this case it is *restore* and *collapse*.

3.4. Event Multicaster

The multicaster is very similar as well. Just as in the e-mail example, you must create the *add* and *remove* methods that accept pane switch listeners as arguments. Then create the *collapse* and *restore* methods. Note that you do not need to create a new multicaster for each event class you create. You may choose to have a single multicaster class for all of your widgets. I have chosen to combine the events from both examples into one multicaster class. See Listing 4 for details.

3.4. Hooking it Up

Finally, we need to complete the WindowBar widget. This widget will simply change its text from “collapse” to “restore” and back again when clicked. In addition it sends the corresponding event. Take a look at Listing 6 to see how it's done.

3.6. Exercises for the Reader

- **A visual window bar**: Create a subclass of WindowBar that renders itself graphically, instead of as a button. Read up on mouse listeners and mouse events; you will need to listen for them.
- **A more complete window bar**: Consider additional actions such as maximize, close, etc. What classes change and in what way?

Example 4. Collapsing Pane

The collapsing pane widget is a container containing exactly one component. It provides a window bar across the top, and the contained component takes up the rest of the area. When the bar is clicked, the widget collapses to display just the window bar. When the bar is clicked again, the component is restored. This widget does not need to generate events. However, it does need to listen for events from the window bar and take action based on them.

4.1. Layout

The CollapsingPane class is a subclass of Panel. I install BorderLayout as its layout manager. The component that will be collapsible is installed in the center area, and the window bar is installed in the north.

```
class CollapsingPane extends Panel implements
        SwitchPaneListener {
        public CollapsingPane( Component c ) {
                setLayout( new BorderLayout() ) ;
                WindowBar bar = new WindowBar() ;
                add( "North", bar ) ;
                add( "Center", c ) ;
                bar.addCollapseListener( this ) ;
        }
}
```

4.2. Handle Events

This class does not generate events, but it must handle them. It is listening for switch pane events. When it receives a *collapse* event it must collapse, and restore itself upon reception of the *restore* event. We have seen how to listen for events and trap them (take a look at the **collapse** and **restore** methods in Listing 7). Now, let's look at what to do when we receive the event. In particular, how do you go about collapsing a component? Every component in the AWT can have its visibility set true or false. But simply setting its visibility is not enough; you must also re-compute the layout and redisplay the parent component. Here is how I did it:

```
private void redraw() {
        Component x = _containedComponent ;
         while( x.getParent() != null )
                {
                x = x.getParent() ;
                }
        x.validate() ;
        x.repaint() ;
}
```

This method simply searches up the component tree until it finds the top window (your applet most likely, but it could be a free-floating window or an application frame). Once the topmost window is found, I ask it to validate. This will cause the layout to be re-computed (items that are not visible will not be included) and re-displayed. An example of the collapsing pane widget is shown in Figure 8. This example uses the e-mail entry widget as the component that is collapsed.

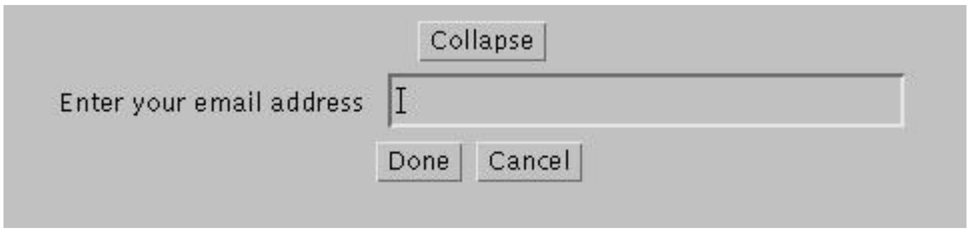


Figure 8. CollapsingPane Widget Used in an Applet

4.3. Exercises for the Reader

Modify the class to accept any arbitrary window bar. You can do this by adding a method to set the window bar or create a new constructor. Remember that in order for the class to the work, the window bar you install must be a subclass of WindowBar. Also, you might think of a way to remove that restriction using interfaces.

Conclusion

We have discussed techniques for designing Java Widgets that are reusable. The resulting widgets are powerful and pluggable enough to warrant the extra effort involved. This article is a starting point, and I encourage you to explore and experiment with new and innovative ideas for your own widgets. I have chosen to focus on the capabilities and design goals. I strongly recommend that you extend these widgets to give them visual “punch”.

You can download all of the examples from this article from *Linux Journal*'s FTP site (see Resources) or from the Harris web site at http://www.hisd.harris.com/Capabilities/java/. At the Harris web site you will be able to see the applets in action and explore some more advanced iterations of similar widgets.

R. J. (Bob) Celestino holds an undergradutate degree in Mechanical Engineering and advanced degrees in Electrical and Computer Engineering. He has been a Linux devotee for more than four years. When not recompiling his kernel or pushing Java to its limits, he enjoys spending time with his wife and three kids. He pays the bills by posing as a software engineer at Harris Corp. in sunny Florida. He can be reached via e-mail at celestinor@acm.org.

Archive Index Issue Table of Contents

Advanced search

Building a Distributed Spreadsheet in Modula-3

John Kominek

Issue #49, May 1998

Mr. Kominek introduces us to the Modula-3 language and shows us how it can be used for cross-platform programming.

Back when Borland introduced Turbo Pascal 1.0, Philip Khan did something shrewd: he included the source code for a simple spreadsheet, which is why many programmers bought the product. At a time when Lotus 1-2-3 was *the* killer application, nothing was more enticing than a glimpse of its key data structure—the sparse matrix.

Of course, the spreadsheet is no longer leading edge. So what might its updated version be? Judging by recent market fanfare, I'd say a spreadsheet that is distributed, multi-platform and web-aware. How would you go about building one?

Delphi, the most recent incarnation of Pascal, is not a bad choice—provided you can live within Windows alone. For us, however, Linux compatibility is a must. You could try to master the intricacies of CORBA, but that standard is now engaged in a turf war with Microsoft's DCOM, a creature of even more convoluted behavior. However, there is another choice available to the Linux programmer.

The Modula-3 language and its surrounding system offer a simple, clean, mature and robust tool for writing distributed applications. (See the sidebar “A Brief Biography”.) In this article I'll highlight the steps necessary for building a distributed spreadsheet. My goal is not to provide a full-fledged product, but rather a framework of code that illustrates all the key components.

A Distributed Application Framework

There are three senses in which a piece of software can be considered “distributed”.

1. The data and computation can be divided into separate processes. In particular, the data can be viewed from multiple clients (GUI viewers), even though it is stored elsewhere.
2. The executables can reside on separate machines—for instance, a pair of Linux servers supporting some mixture of Windows and Linux clients.
3. The work can be distributed between people. You and I may be collaborating remotely on the same spreadsheet, with precautions taken to ensure that I don't overwrite your entries by mistake.

Compared to traditional applications, distributed software is harder to design and get right. In spite of this, it allows for growth and flexible organization.

Software Ingredients

Three basic ingredients are required by our task:

1. **A spreadsheet object**: Initially, it is enough to use a two-dimensional array. Once our application is up and running, experience will help refine the object's interface. Later, the fixed array can be replaced with a sparse matrix.
2. **A display widget**: Having the user interface separate from the data eases modifications and simplifies the task of cross-platform deployment.
3. **Connecting glue**: The spreadsheet object and display widget need to be able to talk to each other.

In Modula-3, *Network Objects* provide the connecting glue. The beauty is that as far as your code is concerned, invoking an object somewhere on the Net is nearly as easy as one inside your own program. Most of the hard work is done for you.

About Modula-3

As a modern, general purpose systems programming language, Modula-3 is lean in design, yet practical and powerful. Applications range from the fun things (multiuser games), to the serious (operating systems), to the deadly serious (911 call centers). Ten years of use has made the reference compiler solid and dependable.

Current implementations exist for Win32 and popular incarnations of Unix. The Linux port, in particular, receives constant attention. Several versions are available for download, including the full source tree. (For pointers, see the sidebar "Modula-3 Resources").

Beyond openness, the language has numerous features to recommend it, including:

- A clean, Algol-derived syntax
- Explicit support for modules and interfaces
- A mechanism for calling external C code and libraries
- Both traditional and object types (with single inheritance)
- Built-in threads and mutexes for multi-threaded programming
- Assertions and exceptions to support error handling
- An incremental garbage collector to simplify memory usage

If this reminds you of Java, that's no accident. Though the syntax of Java is derived from C++, many key improvements descend directly from Modula-3. One implementation of Modula-3 even allows mix-and-match integration with Java.

Features located in "the first ring out", though not defined in the language itself, include:

- Quake, a simplified build language that replaces **make**
- Standard libraries of algorithms and container objects
- A lightweight database component
- A multi-platform windowing system with user interface toolkit
- Network objects

Network objects allow us to proceed in stages. First, a spreadsheet can be constructed as a single executable. Next, as multiple processes running on one machine. Finally, as multiple processes running over multiple machines. The jumps between stages are small.

Step 1: Basic Construction

We need some underlying data structure for our spreadsheet, so let's begin simply by typing:

```
TYPE
  Grid: REF ARRAY OF ARRAY OF INTEGER;
```

or

```
TYPE
  Grid: REF ARRAY OF ARRAY OF Money.T;
```

This defines a two dimensional grid of integers (in the first line), or, as a second option, of type **Money.T**. Integers are a built-in type. **Money.T** is a programmer-defined type; the ".T" suffix is a Modula-3 convention. (In a real spreadsheet, each column would have a distinct user-defined type. Let that detail pass for now.)

A new grid can be allocated on the heap during variable declarations, if you wish, or during program execution.

```
VAR
  myGrid : Grid := NEW (Grid, rows, cols);
BEGIN
  myGrid := NEW (Grid, 100, 20);
END.
```

The second assignment of myGrid will wipe out the first, but don't be alarmed —we do not have a memory leak. The Modula-3 garbage collector takes care of reclaiming lost memory. This is also true of object variables (no destructors necessary), including objects that allocate memory on remote machines.

To flesh out our spreadsheet object, we next attach some operator methods to the grid. A good place for this is in a separate "interface" file. Listing 1 contains an initial cut at spreadsheet.i3. Our object is now declared to be a Spreadsheet.T type.

The important property of an *interface* is that it contains no executable code whatsoever. That's reserved for ".m3" or module files. The interface does not say how something is computed, merely what it does. This is similar to .h files in C, but is more strict. Only the operations explicitly exposed in an interface— or "exported" to use the jargon—are available for outside use.

(The sharp reader may have noticed that the representation of Grid is exposed in spreadsheet.i3—a bad thing. Modula-3 does allow you to hide details of representation inside implementation files. That would take us into a discussion of opaque types, however, a more advanced topic.)

Step 2: User Interface Design

Modula-3 comes with a multi-platform windowing system called Trestle. Built upon Trestle is a user interface toolkit called VBTkit, and a UI builder, FormsVBT. You may call X directly if you wish (alternatively, the Win32 GDI), but in doing so you lose portability.

A description of your program's user interface is called a "Trestle Form". A form is a textual description of names and values, organized using nested parentheses. Form elements consist of windows, frames, buttons and so on, as well as properties such as color. Listing 2 is a sample form for a popup calculator, as shown in Figure 1.

The important point is that a form is defined in its own file, outside any Modula-3 code. This separation of concerns proves valuable when the user interface designer is a different person from the primary coder. The form does not describe how to construct the interface, merely what it looks like. The FormsVBT library builds it at run time and hooks it into your code.

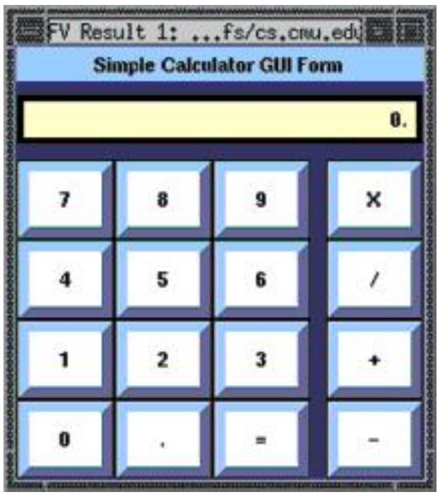


Figure 1. Appearance of Calculator.fv

Step 3: Building the Program

Suppose our spreadsheet is implemented, along with a suite of test functions. To build a program, we must inform the compiler what source files comprise our executable. This is done in a Modula-3 make file, or **m3makefile**. An example is shown in Listing 3.

To build your program, at the command-line prompt type:

```
m3build
```

The compiler will determine dependency relations for you, recompiling only what is necessary.

Step 4: Objects to Network Objects

Converting a regular object (restricted to a single address space) to a network object (visible over the Net) is not as difficult as you might imagine. You must attend to four details.

First, the network object library needs to be linked in. This is performed in the m3makefile (Listing 3).

Second, make the following two changes to the spreadsheet interface:

```
IMPORT Money;
IMPORT NetObj;  (* new statement *)
  TYPE
    T = NetObj.T OBJECT  (* modified line *)
      grid: Grid;
      name: TEXT;
    METHODS
      ...
```

Third, and this matters only at execution time, a network object daemon needs to be running in the background. The program is supplied as part of Modula-3. Start the daemon by typing:

```
netobjd &
```

In a client-server architecture, the spreadsheet object resides with the server, yet it is the client that issues method calls (to update a cell, for example). Clients need to find out about each other. This is the fourth detail.

Step 5: Distributed Deployment

The netobj daemon acts like a bulletin board. First, the server posts a note saying, “I've got a spreadsheet object for sale.” Then the client comes along and says, “I'll buy that.” The server exports; the client imports; the daemon mediates. In the nomenclature of CORBA, the daemon is an object request broker. Once the sale is complete, the client and server talk to each other directly. Code details are found in Listing 4.

Listing 4 will work when the server and client are located on the same machine. Suppose instead that the server runs on some Linux box—eggnog.cmu.edu—and that the clients are elsewhere. Ensure that netobjd is running on eggnog and change one line in the client program.

```
address := NetObj.Locate( "eggnog.cmu.edu" );
```

With that, our programs now talk over the Net.

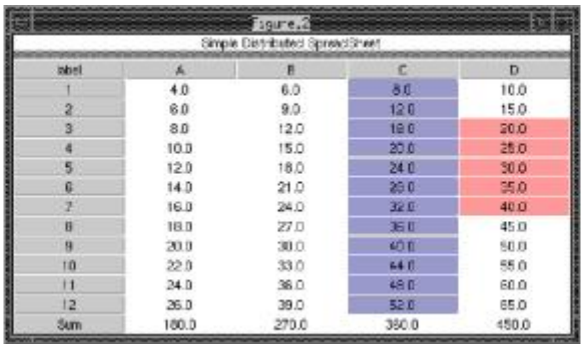
Step 6: Cell Range Locking

Because Modula-3 comes ready-made with thread support, it also provides mutexes (mutual exclusion semaphores) so that parallel operations on the same datum are serialized. In our discussion so far, the Money.T type has been left unspecified. It might actually be something like this:

```
INTERFACE Money;
TYPE
  T = MUTEX OBJECT
    cents: INTEGER;
  END;
END Money.
```

Mutexes protect data so that client B does not modify values before client A is finished. Granted, protecting each cell separately is overkill. A more elegant approach is to protect ranges of cells, with the lock initiated by user action.

Figure 2 shows a spreadsheet from the point of view of user A (Alice). She is working on the cell range tinted red. User B (Bob) cannot modify these cells. He is working on the blue cells, indicating to Alice that to her they are read only.



label	A	B	C	D
1	4.0	6.0	8.0	10.0
2	6.0	9.0	12.0	15.0
3	8.0	12.0	16.0	20.0
4	10.0	15.0	20.0	25.0
5	12.0	18.0	24.0	30.0
6	14.0	21.0	28.0	35.0
7	16.0	24.0	32.0	40.0
8	18.0	27.0	36.0	45.0
9	20.0	30.0	40.0	50.0
10	22.0	33.0	44.0	55.0
11	24.0	36.0	48.0	60.0
12	26.0	39.0	52.0	65.0
Sum	180.0	270.0	360.0	450.0

Figure 2. Simple Multiuser Spreadsheet

Step 7: Porting Procedure

To port our user interface program from Linux to Windows NT, do the following:

1. Archive the client source code by using the **tar** command.
2. Copy the tar file to your Windows machine.
3. Unarchive the file using tar. Convert end-of-line markers.
4. At the command line, type **m3build**.

Assuming there are no stunts of low-level programming, all the Modula-3 code in this example—*including the GUI*—is transparently portable. Differing path name conventions, for example, are hidden behind OS-independent interfaces. There's not an **#ifdef** in sight.

Conclusion

In this article I've highlighted the creation of a multi-platform, distributed spreadsheet using Modula-3. The key step is to wrap the spreadsheet into a network object. In this way, remote objects may be invoked with exactly the same syntax as local objects. Most of the hard work is done for you.

Modula-3 is not the only means for creating distributed applications, but in my mind it strikes an optimal balance between simplicity and power. By its very intent, it is a language for building large, solid systems in order for you to get your work done.

Clearly, my discussion has omitted many details. To help fill this gap, a companion tutorial is available on the Web (see the sidebar “Getting Started”.) Full source code is available for experimentation and invention.

John Kominek holds a master's degree in Computer Science from the University of Waterloo, and is currently a graduate student at CMU. When pressed, he admits to pronouncing Linux to rhyme with Linus. He can be reached via e-mail at jkominek@cs.cmu.edu.

Doubly Linked Lists and the Abstract Data Type

Carl J. Nobile

Issue #49, May 1998

The ADT concept is at the heart of object-oriented programming and cross-platform development. Mr. Nobile gives us an example with his doubly linked list libraries.

Experienced C programmers seeking relief from the drudgery of writing linked lists and dealing with the attending problems of keeping them, somehow, isolated from the rest of their code will appreciate this doubly linked list library. Those who are at an earlier stage in their C programming may also find here a useful tool for enhancing their cross-platform programming skills, as this linked list can serve as an example of an abstract data type (ADT).

What is an Abstract Data Type?

Using ADTs allows the data in a specific piece of code to be hidden from other pieces of code that don't need and shouldn't have access to it. This is often called modular programming or encapsulation. The idea behind the ADT is to hide the actual implementation of the code, leaving only its abstraction visible. In order to do this, one needs to find a clear division between the linked list code and the surrounding code. When the linked list code is removed, what remains is its logical abstraction. This separation makes the code less dependent on any one platform. Thus, programming using the ADT method is usually considered a necessity for cross-platform development as it makes maintenance and management of the application code much easier.

The ADT concept is developed here via the example of how the doubly linked list *Application Programming Interface* (API) was created. The doubly linked list (DLL) can also be an independent C module and compiled into a larger program.

The Doubly Linked List

The DLL package consists of two C modules: dll_main.c comprises the DLL itself and dll_test.c creates an executable program for testing the DLL's functionality. There are also three header files: dll_main.h is included in dll_main.c, linklist.h is included in your application program, and dll_dbg.h is used for debugging the DLL or the DLL's implementation in your application. A word of warning needs to be expressed here: the header dll_dbg.h should never be compiled into a production program, as doing so circumvents the whole concept of ADT programming. The entire package has been compiled on three platforms with four compilers and includes three of the respective Makefiles. Only one of the platforms exhibited any problem because of a compiler that was not fully ANSI compatible. More will be said about platforms later.

Before we get into the philosophy behind this DLL, I want to explain what my goals were when I decided to write this library. It first had to be platform-independent and instantiable; in other words, the DLL had to handle an unlimited number of instances of linked lists in any one or multiple programs concurrently. Also, it had to be robust.

Figure 1. Layout of Doubly Linked List in memory. The arrows indicate to what the Prior and Next pointers point. The Current pointer can point to any Node Struct, so it is open-ended.

In order to fulfill the first requirement, I decided to strictly adhere to the ANSI C standard, and, with the possible exception of how one sets up one's data and uses the DLL's input/output functions, there should be no endian (byte order) problems. The second requirement was met with the creation of a top-level structure. There is only one of these structures per linked list. It keeps track of the node pointers, the size of the applications data in bytes, how many nodes are in the list, whether or not the list has been modified since it was created or loaded into memory, where searching starts from, and what direction a search proceeds in. Figure 1 illustrates how the top-level structure is integrated into the DLL.

```
typedef struct list
  {
  Node      *head;
  Node      *tail;
  Node      *current;
  Node      *saved;
  size_t     infosize;
  unsigned long listsize;
  DLL_Boolean  modified;
  DLL_SrchOrigin search_origin;
  DLL_SrchDir  search_dir;
  } List;
```

This and the next *typedef* structure remain hidden from the application program. The node pointers mentioned above are defined in the next structure, which includes the pointers to the application's data and the pointers to the next and prior nodes. One of these structures is created for each node in the list.

```
typedef struct node
  {
  Info    *info;
  struct node *next;
  struct node *prior;
  } Node;
```

The last definition is a dummy *typedef* of the user's data. It is defined as type *void* so that the DLL's functions will be able to handle any of C's or an application's data types.

```
typedef void Info;
```

As you can see, if the two structures mentioned above are hidden from the application, all of the ugly innards of how the linked list operates will by default be hidden from the application. Thus, we have an abstract data type.

Application Programming Interface

The interface itself follows logically. The first argument of all the DLL's API functions is a pointer of type **List**. This pointer can easily be changed to different lists, thereby accommodating the instantiation requirement of the DLL.

The API's 25 functions are broken down into seven function groups: three initialization, three status, four data modification, six pointer manipulation, six search, two input/output and one miscellaneous. The initialization group handles the creation, initialization and destruction of the DLL. The status group returns various types of information about the DLL. The pointer manipulation group allows the arbitrary repositioning of the current pointer. The data modification group adds and deletes nodes. The search group returns node information based on keyed data fields or on absolute node position. The input/output group saves or retrieves node data to or from a disk file. The miscellaneous group currently only supports version information.

The function prototypes that follow will return two or more **enum** types or the **boolean** type. Some functions have a **void** return value.

```
typedef enum
  {
  DLL_NORMAL,       /* normal operation */
  DLL_MEM_ERROR,    /* malloc error */
  DLL_ZERO_INFO,    /* sizeof(Info) is zero */
  DLL_NULL_LIST,    /* List is NULL */
  DLL_NOT_FOUND,    /* Record not found */
  DLL_OPEN_ERROR,   /* Cannot open file */
  DLL_WRITE_ERROR,  /* File write error */
  DLL_READ_ERROR,   /* File read error */
  DLL_NOT_MODIFIED, /* Unmodified list */
```

```
     DLL_NULL_FUNCTION /* NULL function pointer */
     } DLL_Return;
```

```
  typedef enum
    {
    DLL_FALSE,
    DLL_TRUE,
    } DLL_Boolean;
```

What follows is a short description of all the functions in the API. It is not possible to describe all the intricacies of how the functions are called and what they each return in a short article like this. For this information, refer to the documentation and source code in the distribution.

Initialization

First we need to create a list pointer.

```
  List *listname = NULL;
```

To create the top-level structure, execute the following function:

```
  List *DLL_CreateList(List **name);
```

After this structure is created it needs to be initialized using the next function:

```
  DLL_Return DLL_InitializeList(List *list,
   size_t infosize);
```

That's it—one instance of the DLL is ready to work with; however, there is one last function in this group that is used when we want to permanently remove the list and the top-level structure.

```
  void DLL_DestroyList(List **name);
```

Notice the pointer to pointer notation again; this is used so that **name** can be returned as a **NULL** pointer. The C standard function **free** does not set the pointer; it is passed to **NULL** after deallocating its memory. This can cause a possible problem if that pointer should unwittingly be reused.

I've written a template (see Listing 1) of the initialization sequence. This and the source code in the distribution should help in using the DLL.

Status

The next function tests pointers in the top-level structure to determine if there are any nodes in a list.

```
  DLL_Boolean DLL_IsListEmpty(List *list);
```

The inverse of this function, which follows, creates a new node to see if there is enough memory for a new node. If there is sufficient memory, the temporary node is freed.

```
DLL_Boolean DLL_IsListFull(List *list);
```

To get the number of nodes (records) in the list use this next function.

```
unsigned long DLL_GetNumberOfRecords(List *list);
```

Data Modification

The process of adding new nodes to the linked list can be as easy or as complex as you desire. The following function has the ability to do an insertion sort as it adds nodes or just stick the nodes on the end. Don't let the list of arguments scare you; the function prototyping makes it look worse than it really is.

```
DLL_Return DLL_AddRecord(List *list, Info *info,
 int (*pFun)(Info *, Info *));
```

The first argument is a pointer to the top-level structure, which is the same in all the functions. The second argument is a pointer to the data you want to put into the linked list. The third and last argument points to an optional function that you could write, which determines the sort criteria.

It is worth reviewing how this function should be written, as it shows up again in two other functions described below. It emulates the way the C standard function **strcmp** returns its value. As a matter of fact, it can be just that.

```
int compare(Info *newnode, Info *keylist)
  {
  return(strcmp(newnode->key_element,
  keylist->key_element));
  }
```

Updating the current node (record) is a must in any linked list implementation, and this DLL API is no exception.

```
DLL_Return DLL_UpdateCurrentRecord(List *list,
 Info *record);
```

We would also want to delete the current record.

```
DLL_Return DLL_DeleteCurrentRecord(List *list);
```

The last function in this group deletes the entire list but not the top-level structure.

```
DLL_Return DLL_DeleteEntireList(List *list);
```

Pointer Manipulation

As shown above, there are four pointers in the top-level structure. We concern ourselves here with the *current* pointer. This pointer is where all the power in the DLL comes from and is used in many of the DLL's functions to determine what to work on.

The next two functions move the current pointer to the head or tail of the list.

```
DLL_Return DLL_CurrentPointerToHead(List *list);
DLL_Return DLL_CurrentPointerToTail(List *list);
```

Often, incrementing or decrementing the pointer is necessary:

```
DLL_Return DLL_IncrementCurrentPointer(List *list);
DLL_Return DLL_DecrementCurrentPointer(List *list);
```

It is sometimes desirable to store the *current* pointer, then do something else, and then restore the pointer. We take care of this in the next two functions.

```
DLL_Return DLL_StoreCurrentPointer(List *list);
```

```
DLL_Return DLL_RestoreCurrentPointer(List *list);
```

Search

There is little use having a linked list if you cannot find what has been stored in it. The following functions let you find your data and specify exactly how that data will be found.

```
DLL_Return DLL_FindRecord(List *list, Info *record,
        Info *match, int (*pFun)(Info *, Info *));
```

The first argument, as usual, is the pointer to the linked list. The second is a pointer to the returned data. The third is a pointer to the matching criteria, and the last argument is a pointer to the **compare** function that was previously described in the data modification group. This compare function can be constructed differently, but the idea is the same.

Now life gets a little difficult. The above function needs to know how to look for the data in the linked list. Does it look down from the head pointer, up from the tail pointer, or up or down from the current pointer? My solution to this problem was to use a state table.

There are two more **typedef** enumerations needed, relating to the state table, one to set the origin of the search and the other to set its direction.

```
typedef enum
  {
  DLL_ORIGIN_DEFAULT,  /* Use current origin
```

```
                           * setting */
     DLL_HEAD,    /* Set origin to head pointer */
     DLL_CURRENT, /* Set origin to current pointer */
     DLL_TAIL     /* Set origin to tail pointer */
     } DLL_SrchOrigin;
   typedef enum
     {
     DLL_DIRECTION_DEFAULT, /* Use current direction
                             * setting */
     DLL_DOWN,    /* Set direction to down */
     DLL_UP,      /* Set direction to up */
     } DLL_SrchDir;
```

The state table defaults at initialization to **DLL_HEAD** and **DLL_DOWN**. The **DLL_FindRecord** function uses these values if not changed. To change the operation of this function, use the next two functions shown. If no change is desired in either of these two functions, use **DLL_ORIGIN_DEFAULT** or **DLL_DIRECTION_DEFAULT**. The first function sets the table to new values:

```
   DLL_Return DLL_SetSearchModes(List *list,
           DLL_SrchOrigin origin, DLL_SrchDir dir);
```

The second function returns a pointer of a copy of the state table to the following structure:

```
   typedef struct search_modes
     {
     DLL_SrchOrigin search_origin;
     DLL_SrchDir  search_dir;
     } DLL_SearchModes;
```

The purpose of this function is to check how a succeeding search will be conducted by interrogating the state table.

```
   DLL_Return DLL_GetSearchModes(List *list);
```

Last in this group are three functions that return data relative to the location of the current pointer. They are not affected by the state table.

```
   DLL_Return DLL_GetCurrentRecord(List *list,
           Info *record);
   DLL_Return DLL_GetPriorRecord(List *list,
           Info *record);
   DLL_Return DLL_GetNextRecord(List *list,
           Info *record);
```

Input/Output

Generally, input and output functions would not be considered a part of a linked list implementation; however, they do make life a bit easier when using ADTs. Without these functions one would have to set the current pointer to the head or tail of the list and then make repeated calls to one of the DLL_Get functions mentioned above. If sorting during this process were needed, the task would be even more tedious.

Writing to or reading from a disk tends to be very platform specific. I have striven to make the next two functions as generic as possible; they open files in binary mode and write or read the **Info** structure from beginning to end.

Depending on how you enter data in the **Info** structure will determine if there will be any endian problems.

To save a list, determine the full path to the file, then pass its pointer to the next function. There are no sorting options with this function, because the list is presumably sorted in memory and will be saved in that order.

```
DLL_Return DLL_SaveList(List *list,
        const char *path);
```

When loading a file from disk, you have the option of sorting the list as it comes into memory. Passing a **NULL** loads the file as it exists on the disk and loads it faster than if the list is sorted.

```
DLL_Return DLL_LoadList(List *list,
   const char *path, int (*pFun)(Info *, Info *));
```

Miscellaneous

This last group has only one function in that it returns version information, so a program can determine if it is linking to a different version and check for any incompatibilities.

```
char *DLL_Version(void);
```

What Use is It?

The short answer is it is used for just about any type of data storage where you don't know how much data is to be stored. One example that I've been working with is 3D graphics data where there could be an unknown number of objects in a scene. I've written bar code scanning software that uses this DLL to keep track of all the hand-held terminals that are in use. I also worked on a database conversion program that reads data into one linked list, allowing you to edit it; it then converts the data to another linked list and writes it out again.

Compiling

I'll mainly concentrate on compiling the Linux version; however, there are two Makefiles for DOS: one that compiles using the DJGPP GNU compiler and the other for the MS6.0 compiler. All three Makefiles are included in the distribution. If anyone is interested, there is also a slightly modified version of the DLL that compiles on Big Blues 4690 OS (FlexOS) using the Metaware C compiler (this OS is used in point-of-sale systems).

First, we need to use **tar** to extract the files into the directory where you want it to reside.

```
tar -xvzf linklist.1.0.0.tar.gz -C /your/path
```

The tar file will create a directory named linklist and put everything in it. Next, use **cd** to move to the linklist directory and type one of the following, assuming you're using the GNU compiler:

```
make
```

creates a shared library, or

```
make static
```

creates a static library.

To install the library in the /usr/local/lib directory, enter either **make install** or **make install-static**.

That's all there is to it. You're now ready to write some code.

Conclusion

The concept of the ADT is at the core of object-oriented programming and, as mentioned previously, central to cross- platform development. My linked list example should be of use as either a practical or a learning tool.

Carl J. Nobile currently writes point of sale software and is the administrator of an AIX Unix system for Genovese Drug Stores in New York. At home he is working on a program that can be used to design geodesic homes using ideas from Buckminster Fuller's Synergetics. He can be reached electronically at cnobile@suffolk.lib.ny.us.

Archive Index Issue Table of Contents

Advanced search

The Importance of the GUI in Cross Platform Development

Michael Babcock

Issue #49, May 1998

The fragmentation of development energy into too many GUI toolkits is one of the most serious problems facing the Linux community today.

The key question in making your program cross platform is how to port the GUI. One solution is to write separate GUI front-ends for each platform. This gives you great flexibility and lets you hand-fit each interface to the target system, but soon you find you are rewriting the same interface ideas from scratch. You start to wonder if you shouldn't abstract out some concepts, such as creating a button or drawing a line, and use that abstraction instead. This is precisely what portable GUI toolkits already do. So instead of re-inventing the wheel, or, widget, you will probably want to choose a GUI toolkit and hopefully even improve it.

Wrapper vs. Emulated

There are two approaches to providing platform-independent GUI functionality. The wrapper approach *wraps* the native system widgets in an abstraction layer that provides a common functionality among the different systems. The emulated, or *pure*, approach simply intercepts the native drawing calls and then uses those calls to implement its own widgets.

Wrappers are easier to program because you don't have to write your own widgets. The look and feel of the native target platform is easily maintained because, under the wrapper, you are using the native system widgets. However, wrappers also lose flexibility because they can only provide what the native widgets already provide—a “lowest common denominator” approach to portability. They can't be extended. They do not allow you to exploit the full power of a toolkit.

I personally prefer the *pure* or *emulated* widgets over the native wrapper classes. If you are impressed with certain native widget features in the latest release of Windows, you may fear that emulated widgets will lag behind those native, wrapped widgets. For me, it's the other way around. I see the emulated approach as having the flexibility to exceed native platform features. I believe that if the majority of free GUI toolkit makers were working on a common “Linux GUI API”, we would soon outpace Windows and other company-controlled GUI development with the excellence of our new and emulated widgets.

I'm working on programs that need to be fully multilingual, including fairly complicated composition of Chinese, Japanese and Korean characters (kanji). I'm not impressed with the new native widget features of Windows; they don't come close to doing what I need. So with widget wrappers I'm stuck basically rewriting every widget from scratch based on a Canvas widget, whereas with *pure* widgets I can use the usual object-oriented techniques to extend existing widgets (assuming the API is well-designed). This provides greater flexibility and consistency when writing GUI applications.

For Linux and UNIX there is another problem with native widget wrapping; it is not clear what constitutes native widgets. In Windows and Macintosh this is obvious. But there is no standard GUI API for UNIX (other than the X protocol, which is not a full GUI). The closest UNIX gets is Motif, which is not really a standard and is not free. In other words, in the Linux world, even before you consider questions of portability, you first must choose a GUI library. You must determine what widgets will be your native widgets. In addition to providing all the widgets and structure you want for Linux programs, your chosen GUI toolkit may offer assistance in cross-platform development. And that is what you want. Programming is now more efficient, as you only have to master one set of tools.

The Programming Language Interface

Since C and C++ are so popular, most of the toolkits have C or C++ interfaces. In fact, most of the time it is C++ because object-oriented programming seems particularly applicable to GUI code. However, some, such as Fresco, attempt to be language neutral and potentially provide an interface to almost any language. These toolkits come as libraries that must be compiled and linked for each target system. On the other hand, there are the interpreted languages such as Smalltalk, Tcl and Java that can run on multiple systems without having to be compiled for each one. Then there are the toolkits written in C, such as GTK, that can be called from many other languages such as Scheme, Python or Perl. (Note that GTK is not currently cross platform, but see below.)

Linux needs a standard GUI API. It's not that all applications must end up looking and even acting alike as in Windows, but they should be consistent in certain areas; for example, a consistent desktop, consistent help system, cut and paste, drag-and-drop and so forth.

The fragmentation of development energy into too many GUI toolkits is one of the most serious problems facing the Linux community today. There is some recognition of the magnitude of the problem but nobody can agree on which GUI toolkits to use. A good example is the Gnome and KDE desktop projects; Gnome uses GTK, and KDE uses QT.

For Windows and Macintosh, it's simple; you don't have a choice. Naturally, I prefer the fecundating chaos of the world of Linux and GNU to the stifling dictatorial conformity of the Microsoft domain but, as a programmer, it would be satisfying if the choice of GUI toolkit were a no-brainer. It would be nice if there were such a thing as the “Linux GUI API” so that the synergy of the Linux developers' community could better foster the creation of innovative and compelling programs rather than merely innovative but unfinished tools.

A Look at What's Out There

In contrast to the well-focused, Linus-centered Linux kernel development, everyone has their own idea for GUI toolkits and proceeds in their own direction. They usually only make it 90% of the way. A new toolkit comes along, reduplicates the first 90%, then fades away or is overtaken by another toolkit that charges ahead but never reaches the goal. These GUI toolkit development projects cannot seem to sustain the energy and support needed to add that last 10%. The contents of this “last 10%” vary, but usually include such things as support for internationalized input methods and fonts, and threads. This is frustrating, to say the least.

If I had my way, Fresco would be the number one toolkit for everyone to work on. However, I can't force anyone to do that, and I can't do it all myself. Many others, with different favorites, are in a similar predicament.

Not everyone will share my taste (understatement), but here are my thumbnails of some of the most promising candidates. Note that most of my own interest in cross-platform development has been for Linux (UNIX/X11) and Windows. If a solution also supports Macintosh and OS/2, it's merely an extra bonus. I have favored toolkits that are free and unencumbered of odious licensing restrictions. There are about 100 GUI Toolkits listed on the “GUI Toolkit, Framework Page”, so clearly this is just a small sampling.

Fresco

Fresco is an object-oriented user-interface toolkit, designed by Mark Linton, among others. Mark Linton led the development of the InterViews toolkit at

Stanford, and he seems to have learned a lot about GUI API design from that experience. Fresco is implemented in C++ (Ada and Java versions are also available) but the programming interface is written in IDL (Interface Definition Language) and so it is language neutral. Fresco uses CORBA and supports distributed graphical embedding. It supports structured graphics and resolution independence. It supports X, Windows and Macintosh. Widget implementations are exposed through *kits* which can allow a different look and feel on each platform; however, only a Motif look and feel has been implemented.

With Fresco it is very easy to extend widgets and write new ones. I wrote a simple, yet flexible table widget in a couple of hours. It uses the concept of a DAG (directed acyclic graph) of glyphs, which can be figures, layouts or widgets. The DAG includes full 2-D transformation information, which makes for some impressive demos. One example is a drawing editor that embeds a copy of itself in the drawing, scaled and rotated, that is still fully functional. It is both cool and powerful. For layouts it uses TeX's concept of boxes and glue, which is also convenient and powerful.

However, I guess Fresco is one of those things that is almost too good to be true. Originally a contender as a new X consortium standard, the companies that sponsored the work have abandoned it, and commercial UNIX is going with Motif instead. This places a lot of responsibility on anyone who decides to use Fresco. If there is anything you need, you'll probably have to write it yourself. The two biggest missing pieces are support for printing and for the system clipboard/selection. Fresco needs more widgets, more polishing, and more documentation, i.e., it needs more people working with it and on it.

I recommend looking at the design of Fresco even if you don't plan on using it. Fresco may just be ahead of its time. Look for some of the concepts included in Fresco to be hailed as brand new revolutionary technology in the next 10 years.

OpenStep/GNUstep

OpenStep is a GUI API (along with some non-GUI functions) originally based on the NextStep system. GNUstep is a free implementation of OpenStep. It is a work in progress and has not really reached a usable state yet.

The API seems above average. One nice feature is the use of the powerful Display Postscript for screen drawing which makes printing close to trivial.

The biggest issue is that it uses Objective-C. I don't feel like learning one more language, but after years of trying to find the best solution to the GUI toolkit issue, I may. With the recent Apple-Next hype going on, OpenStep could get enough momentum and hype behind it that using it for free software could pay off. Apple has adopted OpenStep as the future direction for the MAC API, and they will also provide an implementation for Windows. However, the future success of Apple and OpenStep is unclear.

GNUStep is licensed under the LGPL so it can be used in both free and commercial programs.

wxWindows

wxWindows was one of the first free C++ cross-platform GUI toolkits. It takes the wrapper approach to providing cross-platform functionality, wrapping either Motif or XView on X and Windows widgets on Windows. There is also a partial port of an older version of wxWindows to Macintosh.

wxWindows has much practical functionality and lots of development activity, but it suffers from the disadvantages of wrapping other widget libraries rather than implementing its own. It's not as consistent and flexible as it could be. Different platforms often have different functionality and different bugs, and development between platforms is not always in sync; therefore, you have to be careful that your program actually works on all platforms.

There are many extra high-level widgets written in wxWindows, such as tables and HTML viewers. wxWindows also provides some higher-level architecture functionality such as a Doc/View and Print Preview, although I have not personally tested these.

If your program doesn't require more advanced interfaces than provided by the common functionality of the simple native widgets across platforms, wxWindows is a good, practical choice.

A new version of wxWindows in the works, version 2.0, will have some changes in the API in an attempt to make it more functional. There is also a project to make wxWindows wrap the GTK widget library, and a project to implement *generic* widgets using wxWindows drawing calls. It is too early to predict how these changes and additions will affect the future of wxWindows.

Tcl/Tk

Tcl/Tk is a popular solution for GUI programs on X, and it has been ported to Windows and Macintosh. Tcl is a simple scripting language, and Tk is a widget set that can be used with Tcl to create interfaces. I do not find the Tcl language particularly appealing, and Tk is tied fairly closely to Tcl, although some effort is being made to more cleanly separate the two. There are bindings for other languages such as Scheme, Python and Perl. However, using Tk from C or C++ is reportedly somewhat awkward. I have noticed that Tk applications tend to be rather sluggish, but I don't know if this is because of Tcl or the Tk widgets themselves.

One other disadvantage of Tk is that the look and feel (sort of like Motif) is the same across all platforms, so the interface may look out of place to Windows and Macintosh users, although I have heard attempts to remedy this are underway.

Despite the disadvantages, Tk does have a lot of full featured widgets. I understand it is possible to create interfaces relatively quickly. It is certainly worth considering.

Java

A programming language and portable virtual machine and a collection of libraries (called packages), these three technologies together are now apparently called “Java”. Java has received a lot of hype in the past couple of years. While the virtual machine and the rigidly specified language provide some minor portability features, the most interesting part of Java to me is the cross-platform GUI API. The original GUI API, known as the AWT, is a simple wrapper library that has nothing in particular going for it. However, Sun is now creating a new set of pure Java widgets, known as “Swing”, which seems to be well designed and fully featured. With all the hype and momentum behind Java, Swing has the potential to become one of the best GUI libraries available.

The disadvantage, of course, is that Swing (part of the JFC, Java Foundation Classes) is about as far from language neutral as possible. If you want to use Swing, you must take the Java language and the other Java libraries with it, generally abandoning your perfectly good existing libraries.

My biggest complaint about Java is just that I feel like I'm not really developing for Linux anymore; instead I'm developing for the “Java platform”. I get fatigued wading through all the hype and nonsense that the trendiness of Java engenders, and I miss the refreshing honesty of the Linux world. I'm also not totally comfortable with the fact that Sun controls the direction of Java. If we in the free software world don't like something about it, there's ultimately nothing we can do, despite Sun's assurances of *openness*. There are free implementations of the Java language and virtual machine, but at the rate Sun is creating APIs, free implementations of the libraries trail far behind.

I would like to see the ability to use the high quality Java JFC library and still integrate with the direction of the free software world. Perhaps some cooperation with the GNOME project to allow Java applications using Swing to comply with the GNOME application policies would be helpful. Then people could write GNOME applications in Java, even if they used JFC instead of GTK.

Qt

Qt is a commercial C++ toolkit available for X and Windows. It is not free in the monetary sense, costing about $2200 for both the X and Windows version. There is a special exception: if you write a *free* program for X you can use it for free. However, this free program is not really free in the GNU sense, or in the Debian Free Software Guidelines sense, which causes many people (including me) to be wary of basing projects on Qt.

Technically, Qt is reasonably well designed. Particularly notable is its flexible “signals and slots” method of event handling. Qt is being used in the KDE desktop project.

Win32

Win32 is the API for Windows NT and Windows 95. Because of its popularity, it is also being used as a cross-platform API. Microsoft sells an expensive package that will allow you to compile Win32 programs for Macintosh. There are also expensive Win32 libraries available for various flavors of UNIX. The Wine (windows emulator) project is attempting to create a free implementation of Win32 on top of X, along with a binary emulator to run Windows executables directly.

The problem with using Win32 for Wine is that Wine is not mature enough yet, and because the API is controlled by Microsoft, the free implementation will always lag behind Microsoft's own. Win32 is, obviously, a Windows-centric API, and it is not a particularly good API, so the prospect of using it to develop Linux GUI programs is not very exciting. However, if you already have a lot of Win32 code written, or are already very experienced with the API, it may be worth considering investigating one of the implementations.

Motif

OSF/Motif is a set of commercial libraries and widgets built on top of X Toolkit Intrinsics (Xt) which in turn is built on top of Xlib, the lowest layer of X. Motif, in my opinion, is just adequate. Creating applications with Motif is tedious, and from the user's point of view Motif is also just passable, nothing to get excited about. Unfortunately, it is Motif, rather than something technically excellent like Fresco, that the commercial UNIX vendors have declared the official UNIX GUI standard (along with the Common Desktop Environment, CDE, which is built upon Motif) under the auspices of the Open Group.

However, just being declared a standard doesn't make it so. Many people dislike the mediocre quality of Motif and use other solutions for programming X applications. And since Motif is not free, it is not very widespread among Linux users. The vast majority of X applications included in Linux distributions do not use Motif. So, regardless of the Open Group's decree, Motif cannot really be considered the obvious native UNIX GUI library the way Win32 is for Windows and QuickDraw and the Toolbox is for Macintosh. The best that can be said is that most toolkits for X tend to provide a look that is somewhat similar to Motif.

Motif does have one advantage, though; it does provide the ability to create much of what you will ever need in a GUI for your program, even if it takes a lot of time and effort. Motif has much of the functionality of that last 10%, such as full keyboard control, a resource system to customize widgets, support for internationalized input methods and fonts and for threaded programs.

There is a free implementation of Motif available, called Lesstif, that is just becoming usable for some applications. It still needs work to provide the coverage that the latest version of Motif (2.1) has, however. There are commercial versions of Motif for NT, although they are expensive, so it is possible to use Motif in a cross-platform application. I believe Xlib and Xt have already been ported to NT, and theoretically I suppose Lesstif could be ported, which, again theoretically, could provide a free solution on NT.

GTK

GTK is the GIMP ToolKit (see *LJ* Issue 47), the widget library used in the free image manipulation program the GIMP. (See *LJ*, Issues 43, 44, 45 and 46.) The most interesting thing about it is that it seems to be gaining some momentum in the Linux free software world, as more and more projects are using it. Perhaps most notable is GNOME, a project to create a unified, consistent graphical desktop environment built entirely on free software.

In the software world, momentum is often more important than technical design, so GTK is worth investigating for that reason alone. Not that GTK is technically bad. It is a fairly low-level toolkit, written in C, so it doesn't provide a lot of high-level support. The attempt to use object-oriented design implemented in C creates a lot of busy work in the code that is somewhat distracting. However, because it is written in C, it can be used by almost any language, and there are already bindings for C++, Guile, Scheme, Objective C, Perl and probably others. This is no doubt one of the reasons for GTK's popularity.

The design seems reasonable. It is not as flexible as Fresco, but at least it gets some of the basics right, like having a button contain a widget rather than a character string. It also provides layout using horizontal and vertical boxes, which although I found the methods not as intuitive as the TeX inspired boxes and glue of Fresco, they still provide a reasonably straightforward interface.

For handling events, GTK uses a system of signals and slots, like Qt. The C++ interface to GTK, known as GTK--, also provides a nice implementation of the signal/slot methods using templates, an improvement over Qt's macros.

GTK is still immature. It lacks support for full keyboard control, a resource system, unified printing interface and internationalized input and display. It also is currently only for X. It is implemented on top of a low-level, thin wrapper around some Xlib functions, called GDK. This may make porting to other systems easier, although if the wrapper is so thin it requires Xlib semantics, it may be harder. I include it here because ideally the best Linux GUI toolkit will also be a cross-platform GUI toolkit. I hope that as GTK matures into a more obvious choice for a Linux GUI toolkit it will also become a more obvious choice for a cross-platform solution, and we won't have so much fragmentation and duplication of effort.

In short, I have found no obvious winner among the various toolkits. I'm using Java and the Swing package now, while investigating GTK and others in more detail. Ah, yes, I am still dreaming that Fresco will rise again, Phoenix-like, from the ashes.

Non-GUI Considerations

Resources

Michael Babcock has been using Linux since 1992. His programming interests include multi-lingual software (especially Chinese and Japanese), parsing techniques, graphics and anything that will help improve and promote Linux. He enjoys playing basketball, playing the guitar and listening to The Fall. He expects to graduate from the University of Montana in May 1998 with a bachelor's degree in computer science. He can be reached via e-mail at michael@kanji.com.

Archive Index Issue Table of Contents

Advanced search

Rapid Prototyping with Tcl/Tk

Richard Schwaninger

Issue #49, May 1998

A discussion of rapid prototyping and how it can benefit programmers in creating software to match the customer's needs.

Creating software is a complex process that embeds the programmer in rules and constraints. Customer needs fight against bugs in the program, and usability fights against production costs.

The current procedure to solve all these problems is object-oriented design and analysis, accomplished by methodically trying to split a problem into suitably small subdomains and providing a predefined path from analysis through design to implementation and testing.

Such a procedure may help for large programs and may be necessary if many programmers are involved. Most of the really good programs (such as those available for Linux, and even Linux itself) are written by only a few people, and more importantly, most of them don't start out knowing in which direction their software will evolve. Therefore, another paradigm is needed—one that is better suited to the needs of humans.

Here is a statement about rapid prototyping (RP) I found on the home page of Cycad Corporation at http://www.cycad.com/:

> Rapid prototyping is acquainted with keywords like “ *Get to market faster*, *Beat the competition*, *Maximize profitability*, *Increase quality*...” Not only does rapid prototyping provide all the benefits of getting to market faster, but for the first time it is possible to create Proof-of-Concepts to show customers before implementation and create proto-production units before committing to manufacturing. Both of these validation steps can save extensive amounts of resources and time.

While all of the above is certainly true, RP is particularly suited to the creation of software tailored to the customer's needs. One of the most important aspects in determining these needs is to find out what the customer *really* wants. Because the program takes its final shape while it is under development, not in advance as would happen with standard structured design, the programmer can respond to new information about customer needs. With RP this process is straightforward and allowed, while normally it indicates an analysis or design error in other methodologies.

Equally important is an easy way to test new ideas. The final product will be better if you have some sort of playground to find out before implementation whether or not your bright new concept is really as bright as you think. This is even more important in our fast changing world where we have to react quickly to new or changing demands.

Using rapid prototyping for software development does not necessarily depend on a language specifically designed for it. It is possible to use compiled languages like C/C++, but best results can be obtained with an easy to learn/ easy to use interpreted language. To get the most benefit, turnaround times have to be short and modifications to existing code should be possible without much effort.

One area where rapid prototyping really shines is user-interface design (provided you have the right tools). You can create a dynamic visual model with minimal effort and provide users with a physical representation of key parts before system implementation. You can accommodate new or unexpected user requirements earlier, and modifications are much easier.

Another aspect is quality assurance and control. The same tools used to build the prototype can be adapted to build up a test harness that is useful up to the final implementation of the system. This results in fewer changes after delivery and therefore productivity improves.

Tcl/Tk

One of the current trends in computer languages is the use of portable languages like Java, Perl or Tcl. While these are generally slower than a compiled counterpart they offer a lot of advantages for the developer (portability, simplicity, short turnaround times, etc.). If such a language can integrate compiled code (e.g., by using shared libraries), a developer can first concentrate on creating a working solution and later optimize the code at the critical spots.

Tcl/Tk is especially well-suited to rapid prototyping, as it has solutions to all of the requirements discussed above:

- It's simple to use and to learn.
- No compilation or development overhead is required—just write and run your code.
- Graphical user interfaces can be built with very little effort.
- It can be used for testing (through the use of its introspection facilities).

I do not wish to begin a language war here—the above statements are my own opinions. To make the best use of any language you need to know the language very well (Tcl is no exception) and you need the corresponding tools (syntax sensitive editor, debugger, performance analyzer, testing and documentation tools). Finally, you need a lot of good libraries and extensions which give you the power to create big applications with only a few lines of code.

An Example

The following example is meant to give you a feeling for what rapid prototyping is like. To keep it short and understandable it does not use any of the features of Tk at all. Although a graphical example would certainly be better suited to show the advantages of RP, the concepts presented can easily be extended to make use of Tk's extra features. It is one of my actual projects, and I will show it to you in the same way I developed it. As you may guess the *customer* and the *programmer* are in fact just one person—me.

Customer: “I need a nifty little library tool to work with values that are configurable by the end user. The values should come from a simple text file, and that's all there is to it.”

Programmer: “Okay. This one is really easy.”

I sit down and type some lines of Tcl code that give me the following interface:

```
proc Cfg <
```

This way I can use the configuration interface for a (hypothetical) text editor in this manner:

```
set Lines [Cfg "NumLines"]
```

The format of the file should be kept simple, so I try the following to read values from it:

```
NumLines 20
WordWrap 1
```

Reading the file into memory would need another function. As I don't want to pollute name spaces (true for both Tcl and C), I modify the signature of my procedure to be the following:

```
proc Cfg <
```

which gives me the following:

```
Cfg read "myconfig.cfg"
Cfg get NumLines
```

Code to implement all of this is written in 15 minutes. I add another 15 minutes to write some test cases so I can always verify the correctness of the whole implementation. Here is one test:

```
Test c1 "get simple value" {
  Cfg read "test.cfg"
  Cfg get NumLines
} 20
```

It reads my simple test configuration file and checks if the value for NumLines is really the same as set in the configuration file. It is all finished in 30 minutes. Sure, I have to rewrite it in C, but first I'll cross check with my customer.

Customer: “Nice work, but you see, if an application gets bigger, we may have name clashes—consider two modules that use a configuration value named Width. One could be the width of a window and the other the width of the text in the editor. What we really need is a module dependent solution.”

Programmer: “Grrr. I should have known—simple things tend to get complicated with time.”

Modifying the interface is straightforward, but what about the file format? Far back in my mind is a little hint named win.ini. I cross check with a real windows installation on my DOS partition—yes, the format would be just right for my problem, and it will already be familiar to a lot of users.

So, I modify my configuration file to look like this:

```
; my config test file: 24/02/1997
[Editor]
WordWrap=1
NumLines=20
Width=80
[Window]
Width=320
Height=400
; EOF
```

Reading such a file is far more complicated than the simple approach taken earlier. I pat myself on the shoulder that I have not yet written any C code.

Using some of the extra features of Tclx, an extension to Tcl, the file reader is ready and working in half an hour.

Now, I modify the interface as follows:

```
  Cfg read "myconfig.cfg"
  Cfg get Editor NumLines
  Cfg get Editor Width
  Cfg get Window Width
```

and I modify all the test cases. Again, I present the whole thing to my customer.

Customer: “Yeah, that's just what I wanted. I have some real world examples, could you verify that they work correctly?”

Programmer: The customer gives me some *real* X Window System configuration files—did you think the above were *real world* examples? Anyway, I run some tests on them and poof, all my nice looking code breaks. A quick look at the files reveals the problems—backslashes, brackets, blanks and entries that span multiple lines. Parsing an X configuration file is apparently more complicated than I thought at first.

I add a lot of new test cases that show all the above failures. They help me to make sure that I have a working piece of code at last, without one change breaking something that worked previously.

Adapting the Tcl code is comparatively easy. It takes some time though, as I have to fiddle around a bit with regular expressions for the parser. As soon as all the tests complete without error, I show my work to the customer.

Customer: “While waiting, I have used your code in a real world program, and it has one big drawback. Consider the following example configuration:

```
  lpr -#1 -Plj5 myfile.txt
```

This is a print command which should be executed with **exec**. Some parts are static but depend on the operating system (e.g., lpr versus lp), others (such as the selection of the printer) may be done by the user and still others (such as the file name) come directly from the program. What I need is a flexible substitution scheme for configuration values.”

Programmer: “Ha. Now that's a really big request. How do I create such a *flexible substitution scheme*?”

It takes me a day of thinking (actually this was handled by a background process in my brain) to find *the* solution—Tk uses substitution for its key binding mechanism, so why not use the same ideas here?

Again, I modify the interface, so it can process code such as the following:

```
   Cfg get Printer Command\
      "%c=1" "%f=myfile.txt"
```

and I add some lines to my configuration file:

```
   [Printer]
   Command= lpr "#%c "Plj5 %f
```

Before returning the final value to the caller my routine substitutes all “%?” sequences with the corresponding values from the argument list. Using Tcl's regular expressions makes this quite simple, and the only thing that takes some time is writing the tests to check for all these crazy conditions like “%%” and “\%”.

The above example will return

```
   lpr -#1 -Plj5 myfile.txt
```

just as expected by the customer.

Customer: “Great. This is what I have been looking for for years. There's just one sub-optimal item left. If the user accidentally deletes his configuration file, the program will no longer work. Isn't it possible to keep some default values for this case?”

Programmer: “I have already thought of this situation. The code would be more readable if an actual value were present, and it could be tested more easily as I wouldn't have to write configuration files all the time.”

I modify the interface once more by adding a new subcommand:

```
   Cfg def <module> <name> \
      <default> ?<sub>? ..."
```

The general syntax of the **get** and **def** sub-commands are the same (and both return the same configuration value when called with the same substitution values), but def also sets a default value when none is given.

If a configuration file has been read by Cfg, the values in this file take precedence, otherwise the default is used.

Here is an example:

```
   Cfg def Printer Command \
      "lpr -#%c -Plj5 %f" \
      "%c=1" \
      "%f=myfile.txt"
```

This command returns:

```
  lpr -#1 -Plj5 myfile.txt
```

Customer: “I use your code in all of my Tcl applications, and it works like a charm. The only thing I noticed is that the programs are really slow on startup. I suspect this is related to your configuration code. Couldn't you do anything about that?”

Programmer: “Aaargh.”

Back in the configuration business. First, I check some of the customers applications. No wonder he has slow startup—he uses configurations quite excessively, even for language internationalization. I instrument the applications for profiling and create some snapshots for analysis. The resulting diagrams (see Figure 1 and Figure 2) clearly state the two highest CPU-cycle-eaters: reading configuration files and substitution of configuration values.

Figure 1. Tcl Profiler Bar Chart

Figure 2. Tcl Profiler Graph

The interface is comparatively stable now (the customer has already used it in all his programs, he won't change it that easily). I start rewriting some of the code in C. As I know the exact bottlenecks from the analysis, the first two C routines replace the configuration reader and the substitution engine.

Now all of the previously written tests come in handy—I can check if the new code works correctly with nearly no effort. Both of the routines are quite a bit of work as they do some very tricky operations and pointers always point to somewhere unexpected. Sure it's nice to have a debugger for Tcl (see Figure 3), but without a C debugger, the life of a C programmer would be truly difficult.

Figure 3. Tcl Debugger

The new code is worth the price. Speed can be 5 to 10 times that of the original code, and this can mean the difference between a one second startup and a ten second startup.

There is a good deal more needed to make this little example into a full-fledged library, but it should be enough to see the general concepts. Here are the most important points to learn about the advantages of RP:

- Work can begin with incomplete specifications.
- Different implementations can be created in a straightforward and easy way using Tcl, until the customer is satisfied.

- Working examples can be presented long before the application is complete.
- If speed is a problem, the relevant parts can be rewritten in C.
- Quality increases with added test procedures.

I did a really big project that integrated CAD (computer aided drafting) and PPC (production planning and control) software, and I successfully used the ideas presented above. (See Figure 4 and Figure 5.) The project took more than a year, and the specification changed more than once. The final solution shed another light on the versatility of Tcl: the user is able to define his own components (window frames, automatic actuators, etc.) using Tcl and its object-oriented extensions. This feature was just an offspring of our rapid prototyping philosophy as we simply provided the customer with our own tools.

Figure 4. PPC Software Screen

Figure 5. A Second PPC Software Screen

Using Tcl for software development is not the ultimate bells and whistles solution. There are drawbacks such as the lack of real data types (anything is a string), the inefficiency of interpreted code (although a compiler is now available), and the fact that **eval** stays a mystery even to Tcl fanatics. Tcl shines when your program is mainly centered on a graphical user interface. Other tasks may be done better by Perl, C or even FORTRAN. As dynamically loaded, shared libraries are now quite common on several platforms, we can split problems into chunks and solve these chunks with the language that is most appropriate for it. Tcl/Tk may be the glue for the graphical interface in this scenario.

What is Tcl/Tk?

What is Rapid Prototyping?

Richard Schwaninger (risc@finwds01.tu-graz.ac.at) (risc@ping.at) creates software for product automation systems. He uses Linux as his main development platform and builds tools for Tcl/Tk. He has been in the software business since the days of CP/M and has done a lot of late night hacking with

AutoCad Lisp. He is married and lives in Graz, the capital of Styria/Austria, and he likes outdoor activities such as climbing, skiing and volleyball.

Archive Index Issue Table of Contents

Advanced search

CDE Plug-and-Play

George Kraft IV

Issue #49, May 1998

A major strength of the Common Desktop Evnironment is its programming infrastructure, for example, ToolTalk. This article illustrates client and server plug-and-play through the use of the Desktop's Application Programming Interfaces (APIs).

ToolTalk, in the Common Desktop Environment (CDE), is a message brokering system that enables applications to communicate with each other without having direct knowledge of one another. Client and server applications can be developed independently, mixed and matched, and upgraded separately through plug-and-play. In addition, the Desktop Service can be called to perform methods on file and buffer objects on behalf of ToolTalk.

Figure 1 shows the ToolTalk Service listening for TtChmod client requests. ToolTalk Service brokers pattern-matched Chmod messages to the registered mock change-mode application server (**ttchmodd**) that is waiting to handle the incoming messages.

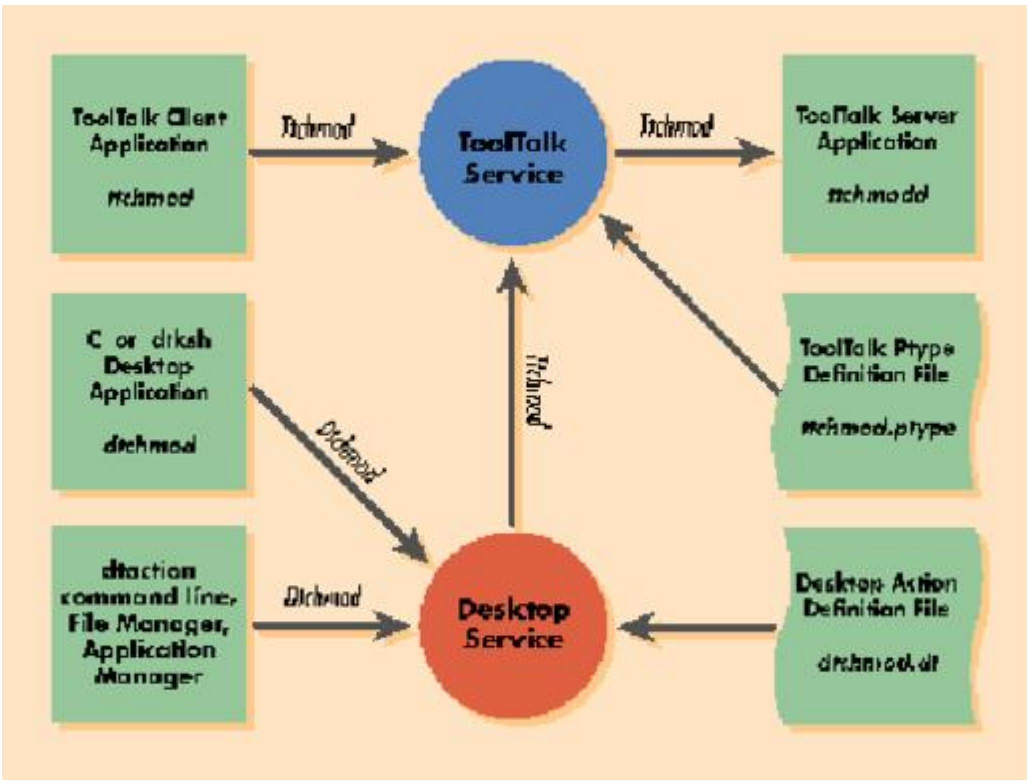

ToolTalk Client Application
ttchmod
TtChmod
ToolTalk Service
TtChmod
ToolTalk Server Application
ttchmodd
C or dtksh Desktop Application
dtchmod
DtChmod
TtChmod
ToolTalk Ptype Definition File
ttchmod.ptype
dtaction command line, File Manager, Application Manager
DtChmod
Desktop Service
Desktop Action Definition File
dtchmod.dt

Figure 1.

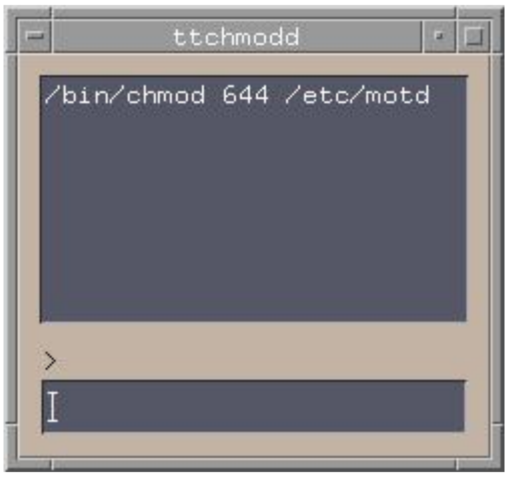
ToolTalk brokers the requests from the client to the server application. The Desktop Service can forward CDE object method invocations to the ToolTalk Service. With the Desktop, both C programs and **dtksh** scripts can initiate actions that are transmitted to the ToolTalk Service. Consequently, client invocations from the **dtaction** command line, application manager icons, and file manager icons can be directed through the Desktop Service to ToolTalk application services. Therefore, double clicking on a file icon in the file manager can be plugged into a ToolTalk registered application by first routing through the Desktop action and data-type service.

Ptype

The key to the ToolTalk message brokering system is its ability to define process-type identifiers with specific operations and arguments. In Listing 1, the process-type (ptype) TtChmod will execute the ToolTalk change-mode daemon application ttchmodd. This occurs when the session operation Chmod with file name and mode arguments are matched from a request. Compiling the ptype definition with the **tt_type_comp** utility will register services for ToolTalk client applications to call. Consider the ptype as a C header file describing an application programming interface (API) and the compiled suite of ToolTalk Services definitions as a library of methods to call. For the list of installed process-type identifiers, try running **tt_type_comp -P** at the command line to dump the database to the screen.

Chmod Service

The file change-mode application (**ttchmod**) described in Listing 2 is simply the Motif command widget. In Figure 2, the ttchmodd server application graphically prompts the user for a command, then calls the callback command **callCB** to execute it; however, for this example, the application just prints the command. The file change-mode application quickly becomes a plug-and-play service when it registers itself with the ToolTalk Service, then listens for messages to be handled by its ToolTalkCB receiver routine.



```
/bin/chmod 644 /etc/motd
```

```
>
```

Figure 2. ttchmodd

Register ToolTalk Service

An application must first locate the ToolTalk session associated with the X display to register itself as a service, as shown in Listing 3. After the application sets its default session to the display session, the application can initiate itself as a ToolTalk process and obtain a ToolTalk file descriptor. When the ttchmodd application gets a handle on the ToolTalk session, then it can register the TtChmod process type and join the session to listen for requests.

Handle ToolTalk Requests

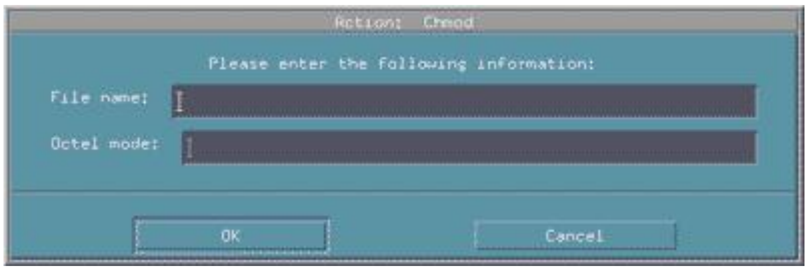
ToolTalk sends a message to a registered service; the service listens on its ToolTalk file descriptor for input. When input is observed, the ToolTalkCB routine is triggered, and the message is read and analyzed. (See Listing 4.) The message's operation is checked, then the arguments are read from the message. ToolTalk messages are similar to a reentrant version of an ordinary C program's command-line arguments. The ttchmodd service is no longer needed after it reads the message, so the recipient tells the ToolTalk service to discard the message.

ToolTalk Client

The ttchmod ToolTalk client (see Listing 5) is much simpler than the ttchmodd ToolTalk server. The client opens a ToolTalk process and locates the session. The TtChmod process-type message request consists of the Chmod operation with the file name and mode input arguments. It is sent to the ToolTalk Service to be brokered to a registered server application accepting the ptype pattern. However, note that this example omits error checking and garbage collection with **tt_mark** and **tt_release**.

Desktop Action

The Desktop Action database in CDE describes methods and objects for applications to act upon. CDE's Desktop Service can describe an action like DtChmod, shown in Listing 6, that can be forwarded through the ToolTalk Service to ttchmodd. If the action does not receive the appropriate arguments, then the Desktop can prompt the user, as shown in Figure 3.


Action: Chmod

Please enter the following information:

File name:

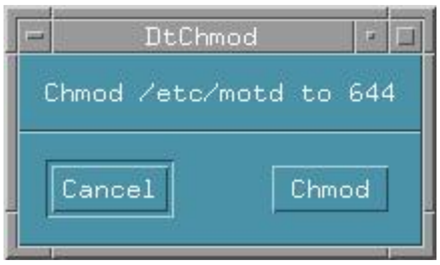
Octel mode:

Figure 3. dtaction

The relationship of the Desktop Action definitions to CDE methods is similar to the relationship of ptype definitions to ToolTalk processes. For an example of Desktop actions and data types, run **dttypes** at the command line to dump the database to the screen.

Desktop Service Client

The APIs of the Desktop Service can invoke actions registered in the Desktop database either from a C program or from a dtksh script, as shown in Listing 7. The dtchmod.ds dtksh script prompts the user, with the message dialogue, as shown in Figure 4, to confirm with the user before requesting changes to the file's mode.


DtChmod

Chmod /etc/motd to 644

Figure 4. dtchmod

In addition to calling Desktop actions from C programs and dtksh shell scripts, users can initiate requests from the command line, as shown here:

```
dtaction DtChmod /etc/motd 644
```

If the appropriate arguments are given, then the action is forwarded to ToolTalk; otherwise, the user is first queried, as shown in Figure 3.

Plug and Play

We have seen how the ttchmodd service registered with ToolTalk can receive messages matching the TtChmod ptype pattern from ToolTalk clients, from Desktop clients written in either C or dtksh, from the command line and from double clicking on file object icons. These examples demonstrate how client and server applications can be developed independently, mixed and matched, and upgraded separately through plug-and-play. A ToolTalk-enabled application service registered with its ptype definition can be developed without specific knowledge of its counterpart.

CDE defines a message dictionary of desktop-specific ToolTalk process types, operations and arguments as seen from viewing the database. Others, such as Computer-Aided Design (CAD) and Electronic Design Automation (EDA) services have developed supplemental dictionaries. You can use existing ptypes or define your own, but the important point to know is how to register the process-type identifier, operation and arguments.

Resources

George Kraft is an Advisory Software Engineer for IBM's Network Computer Division. He has previously worked on CDE V2.1 and V1.0 for IBM's RS/6000 Division and on X and Motif for Texas Instruments' Computer Systems Division. He has a BS in Computer Science and Mathematics from Purdue University. He can be reached via e-mail at gk4@austin.ibm.com.

Archive Index Issue Table of Contents

Advanced search

The Python DB-API

Andrew M. Kuchling

Issue #49, May 1998

A Python SIG has put together a DB-API standard; Mr. Kuchling gives us the details.

Many people use Python because, like other scripting languages, it is a portable, platform-independent and general-purpose language that can perform the same tasks as the database-centric, proprietary 4GL tools supplied by database vendors. Like 4GL tools, Python lets you write programs that access, display and update the information in the database with minimal effort. Unlike many 4GLs, Python also gives you a variety of other capabilities, such as parsing HTML, making socket connections and encrypting data.

Possible applications for the Python DB-API include:

- Many web sites construct pages on the fly to display information requested by the user, such as selections from the products offered in a catalog or from the documents available in a library. Doing this requires CGI scripts that can extract the desired information from a database and render it as HTML.
- An Intranet application might use the **Tkinter** module and the DB-API to provide a graphical user interface for browsing through a local database, such as accounts receivable or a customer list.
- Python programs can be used to analyze data by computing statistical properties of the data.
- Python programs can form a testing framework for programs that modify the database, in order to verify that a particular integrity constraint is maintained.

There are lots of commercial and freeware databases available, and most of them provide Structured Query Language (SQL) for retrieving and adding information (see Resources). However, while most databases have SQL in common, the details of how to perform an SQL operation vary. The individuals who wrote the Python database modules invented their own interfaces, and the resulting proliferation of different Python modules caused problems: no two of them were exactly alike, so if you decided to switch to a new database product, you had to rewrite all the code that retrieved and inserted data.

To solve the problem, a Special Interest Group (SIG) for databases was formed. People interested in using Python for a given application form a SIG of their own: they meet on an Internet mailing list, where they discuss the topic, exchange ideas, write code (and debug it) and eventually produce a finished product. (Sounds a lot like the development process for the Linux kernel, doesn't it?)

After some discussion, the Database SIG produced a specification for a consistent interface to relational databases—the DB-API. Thanks to this specification, there's only one interface to learn. Porting code to a different database product is much simpler, often requiring the change of only a few lines.

The database modules written before the Database SIG are still around and don't match the specification—the mSQL module is the most commonly used one. These modules will eventually be rewritten to comply with the DB-API; it's just a matter of the maintainers finding the time to do it.

Relational Databases

A relational database is made up of one or more tables. Each table is divided into columns and rows. A column contains items of a similar type, such as customer IDs or prices, and a row contains a single data item, with a value for each column. A single row is also called a *tuple* or a *relation*, which is where the term “relational database” originates.

For an example database, we'll use a small table designed to track the attendees for a series of seminars. (See Listing 1.) The Seminars table lists the seminars being offered; an example row is (1, Python Programming, 200, 15). Each row contains a unique identifying ID number (1, in this case), the seminar's title (Python Programming), its cost ($200), and the number of places still open (15). The Attendees table lists the name of each attendee, the seminar that he or she wishes to attend and whether the fee has been paid. If someone wants to attend more than one seminar, there will be more than one row with that person's name, with each row having a different seminar number and payment status.

The examples below use the **soliddb** module, which supports accessing SOLID databases from Python. SOLID is a product from Solidtech that was reviewed by Bradley Willson in *LJ*, September, 1997. I'm not trying to cover CGI or Tkinter programming, so only the commands to access the database are presented here, in the same manner as if typed directly into the Python interpreter.

Getting Started

To begin, the program must first import the appropriate Python module for connecting to the database product being used. By convention, all database modules compliant with the Python DB-API have names that end in “db”, e.g., soliddb and oracledb.

The next step is to create an object that represents a database connection. The object has the same name as the module. The information required to open a connection, and its format, varies for different databases. Usually, it includes a user name and password, and some indication of how to find the database server, such as a TCP/IP hostname. If you're using the free trial version of SOLID, UNIX pipes are the only method available to connect to the server, so the code is:

```
>>> import soliddb
>>> db = soliddb.soliddb('UPipe SOLID',
         'amk', 'mypassword')
>>> db
<Solid object at 809bf10>
```

Cursor Objects

Next, you should create a cursor object. A cursor object acts as a handle for a given SQL query; it allows retrieval of one or more rows of the result, until all the matching rows have been processed. For simple applications that do not need more than one query at a time, it's not necessary to use a cursor object because database objects support all the same methods as cursor objects. We'll deliberately use cursor objects in the following example. (For more on beginning SQL, see *At the Forge* by Reuven Lerner in *LJ*, October, 1997.)

Cursor objects provide an **execute()** statement that accepts a string containing an SQL statement to be performed. This, in turn causes the database server to create a set of rows that match the query.

The results are retrieved by calling a method whose name begins with **fetch**, which returns one or more matching rows or “None” if there are no more rows to retrieve. The **fetchone()** method always returns a single row, while **fetchmany()** returns a small number of rows and **fetchall()** returns all the rows that match.

For example, to list all the seminars being offered, do the following:

```
>>> cursor = db.cursor()
>>> # List all the seminars
>>> cursor.execute('select * from Seminars')
>>> cursor.fetchall(
[(4, 'Web Commerce', 300.0, 26),
 (1, 'Python Programming', 200.0, 15),
 (3, 'Socket Programming', 475.0, 7),
 (2, 'Intro to Linux', 100.0, 32),
 ]
```

A row is represented as a tuple, so the first row returned is:

```
(4, 'Web Commerce', 300.0, 26)
```

Notice that the rows aren't returned in sorted order; to do that, the query has to be slightly different (just add **order by ID**). Because they return multiple rows, the fetchmany() and fetchall() methods return a list of tuples. It's also possible to manually iterate through the results using the **fetchone()** method and looping until it returns “None”, as in this example which lists all the attendees for seminar 1:

```
>>> cursor.execute (
        'select * from Attendees where seminar=1')
>>> while (1):
...  attendee = cursor.fetchone()
...  if attendee == None: break
...  print attendee
...
('Albert', 1, 'no')
('Beth', 1, 'yes')
('Elaine', 1, 'yes')
```

SQL also lets you write queries that operate on multiple tables, as in this query, which lists the seminars that Albert will be attending:

```
>>> cursor.execute("""select Seminars.title
...                 from Seminars, Attendees
...         where Attendees.name = 'Albert'
...             and Seminars.ID = Attendees.seminar""")
>>&Gt; cursor.fetchall()
[('Python Programming',), ('Web Commerce',)]
```

Now that we can get information out of the database, it's time to start modifying it by adding new information. Changes are made by using the SQL **insert** and **update** statements. Just like queries, the SQL statement is passed to the execute() method of a cursor object.

Transactions

Before showing how to add information, there's one subtlety to be noted that occurs when a task requires several different SQL commands to complete. Consider adding an attendee to a given seminar. This requires two steps. In one step, a row must be added to the Attendees table giving the person's name, the ID of the seminar they'll be attending and whether or not they've paid. In the other step, the **places_left** value for this seminar should be decreased by one, because there's room for one less person. SQL has no way to combine two commands, so this requires two execute() calls. But what if something happens and the second command isn't executed—perhaps, because the computer crashed, the network died or there was a typo in the Python program? The database is now inconsistent: an attendee has been added, but the **places_left** column for that seminar is now wrong.

Most databases offer *transactions* as a solution for this problem. A transaction is a group of commands: either all of them are executed, or none of them are. Programs can issue several SQL commands as part of a transaction and then *commit* them, (i.e., tell the database to apply all these changes simultaneously). Alternatively, the program can decide that something's wrong and *roll back* the transaction without making the changes.

For databases that support transactions, the Python interface silently starts a transaction when the cursor is created. The **commit()** method commits the updates made using that cursor, and the **rollback()** method discards them. Each method then starts a new transaction. Some databases don't have transactions, but simply apply all changes as they're executed. On these databases, commit() does nothing, but you should still call it in order to be compatible with those databases that do support transactions.

Listing 2 is a Python function that tries to get all this right by committing the transaction once both operations have been performed. Calling this function is simple:

```
addAttendee('George', 4, 'yes')
```

We can verify that the change was performed by checking the listing for seminar #4, and listing its attendees. This produces the following output:

```
Seminars:
4       'Web Commerce'  300.0   25
Attendees:
Albert  4       no
Dale    4       yes
Felix   4       no
George  4       yes
```

Note that this function is still buggy if more than one process or thread tries to execute it at the same time. Database programming can be potentially quite complex.

With this standardized interface, it's not difficult to write all kinds of Python programs that act as easy-to-use front ends to a database.

Resources

Andrew Kuchling works as a web site developer for Magnet Interactive in Washington, D.C. One of his past projects was a sizable commercial site that was implemented using Python on top of an Illustra database. He can be reached via e-mail at akuchling@acm.org.

Archive Index Issue Table of Contents

Advanced search

Toward Greater Portability: A Quixotic View

Ph.D.. Graydon Ekdahl,

Issue #49, May 1998

A fun way to look at the issue of the development of a universal GUI.

Don Quixote: Ahhh, Sancho, think of it! A single GUI interface that would allow me to write my application once and then run it on Windows, OS2 Warp and Linux boxes without changing one line! Maybe even on Alpha and Sun boxes too.

Sancho: But Master, no one can agree on anything. Especially the big embarcaderos. And if they did agree on something like that, it would only be after a hundred years. Or maybe longer. Then I sleep in the earth. May I rest in peace.

> [*Sancho sometimes transmogrifies vocables when he speaks. It is not clear exactly what he means by "embarcaderos". The meaning probably lies somewhere within the semantic turf demarcated by the concepts "entrepreneurs", "big guys in business", "movers and shakers", or the concept of your choice that denotes mercantile power, grandeur and a touch of narcissism.*]

Don Quixote: Have you forgotten the great examples of cooperation? The ones that made computing easier? What about the LIM memory specification? Three giants—your embarcaderos—created one method of accessing memory above one megabyte so that programs could be bigger, faster.

Sancho: Careful, Master. One of the last giants was a windmill.

Don Quixote: Sancho, it *was* a giant. And remember: I am your master.

Sancho: But Master, just *one* example. Once is an exception. Proves nada! Twice is a coincidence. Also proves nada. Three times is a pattern. Give a second example!

Don Quixote: You doubter! You miss the best in life because you look down, not up. You overlook great things to discover only the pebbles in your own shoes.

Sancho: Please, Master, the example. A poor man like me does not go far on words and ideas. Please, something concrete!

Don Quixote: Sancho, Sancho! What about STL, the standard template library? Now we all have containers that can be used with almost any ANSI compiler! We cooperated on that! What about the committee that drafted standards for C++? Knights from all over the industry came together. These are great achievements.

Sancho: All right, Master! STL and ANSI standards are both C++. You get credit for one example only. Nothing more. That's number two. So now you have a coincidence.

Don Quixote: Have I made my point?

Sancho: No!! You give three good examples, or I believe in nothing! Nada! Nada!

Don Quixote: Sancho, Sancho, so pedestrian, so little imagination. I fear for your salvation.

Sancho: Master, the *example*, please!

Don Quixote: Think of Java. Here we have something like the ideal I am thinking of. We write once, we run many. Why? Java is interpreted code. And the interpreter is written for the platform. The application is written for the interpreter. So we run on all platforms.

Sancho: Oh, Master! If Java is interpreted, Java is slower! Slower is not good. Faster is good.

Don Quixote: Exactly! For once you have something right. We need performance *and* portability.

Sancho: But Master! Where is the cooperation? Sun Microsystems created Java.

Don Quixote: You miss the point. Sun did cooperate. They licensed Java to anyone who agreed not to provincialize the language by creating non-portable, local extensions. Most people respected this agreement. Java was not theirs to take over anyway. So Java is portable, mainly.

Sancho: Master, Master! What is this ahead?

> [*Sancho and Don Quixote have just reached the crest of a hill, and looking down the other side, they spy a bridge over a small brook. Next to it stands a sign: "Bridge to Cross". Just as they approach the crossing, a Proprietary Troll jumps up from below the bridge onto the road, barring their way.*]

Proprietary Troll: I control this bridge, and you must pay toll, or you may not cross.

Don Quixote: Sir, I am a Knight, champion of Dulcinea del Toboso and future keeper of the universal GUI interface. Let us pass, sir!

Proprietary Troll: I control this bridge, and you must pay toll. Heed my words, or you will fare badly.

Don Quixote: Indeed! What will I lose?

Proprietary Troll: Market Share, my Knight, market share!

Don Quixote: [*suddenly thoughtful*] I see. And what toll do you desire?

Proprietary Troll: Control!

Don Quixote: Of what? You mean the GUI interface? You want to control that too?

Proprietary Troll: That most of all. That is your toll. Give it up, turn it over to me, or you will not pass. I control the bridge, I control the interface and I control you.

Don Quixote: Not so fast, you runt!

Sancho: Master, Master! You forget yourself!

Don Quixote: You forget your bridge is just one miserable, narrow crossing, that your brook is just a tributary of one great, massive stream whose power sweeps us all along, that you can simply be washed away if you fail to meet the needs of travelers who come this way. You overreach yourself! Say, aren't you a little far from Scandinavia?

Proprietary Troll: [*puffing himself up and then expelling a lot of hot air*] Try to cross without my help. Lose market share. We will see who is swept away!

Don Quixote: Canis culum in tuo naso! We will find another bridge and cross elsewhere. We do not need you to provide a crossing. If need be, we build a bridge of our own.

> [*"Canis culum in tuo naso" is a Latin curse which first occurs in writing in the Old High German period (ca. 850-1050) in a phrase book for travelers which gives Latin phrases and their German equivalents. The Old High German equivalent is: "hundes ars in dine nas".*]

Sancho: [*to himself*] Such nonsense! For windmills he sees giants. Trolls bigger than life. Oh, Lord, give me something small again, a pebble for my shoe!

Don Quixote: Sancho, forget your pebbles! Let us hie ourselves hence and find another crossing.

> [*Sancho and the Don turn about to find a crossing on another tributary of the great stream. Their thoughts return to the interface.*]

Sancho: OK, you gave three examples. Tell me more about your universal gooey face.

Don Quixote: [*grimaces at Sancho*] Sancho, if I am no longer your master, how will you fill your belly? I said Universal GUI interface, not gooey face. Sometimes I don't think you take me seriously.

Sancho: Sorry, Master! Tell me more about your gooey inter...face.

Don Quixote: That's better. First, the universal GUI interface should be a standard that remains the same from platform to platform just like STL or ANSI C++. The details of the implementation should be hidden from the client because they're platform dependent. This interface means that ideally programs can run on multiple platforms without change.

Sancho: What is the gooey interface supposed to do?

Don Quixote: That's easy. The interface should provide a complete set of tools which perform routine window management tasks and offer all the functionality of the provincial GUI interfaces already in place. These tools should be general enough to be very flexible but powerful enough to compete head-on with the provincial GUI interfaces out there and provide programmers with a serious opportunity to write portable code.

Sancho: Great. But what language will you write this in? Spanish? Portuguese?

Don Quixote: C++ of course! It is object-oriented, supports inheritance, and the details of the implementation can easily be hidden from clients. It is also fast and allows access to the hardware.

Sancho: Can you be more specific? What procedures will your universal gooey interface include to manage a window?

Don Quixote: The details are best left to the Knights who convene to create the interface. All they need to do is decide what functionality the interface should include and then examine the provincial interfaces already in place to see if implementing that functionality is practical.

Sancho: Has anyone ever tried to write a universal gooey interface?

Don Quixote: Not that I know of. [*Suddenly the Don is full of himself.*] But there is rustling in the woods, voices from under the earth, whisperings in the breeze. A new age may be at hand. [*The Don thumps his chest triumphantly as though the battle were already won.*]

Sancho: [*To himself*] What nonsense! [*To the Don*] What do you mean? Your words are confused....uhhh...excuse, Sire, I mean confusing. Can you give an example?

Don Quixote: No one has tried to write what I speak of exactly but there are movements in that direction. The X Window System has been ported from Linux to OS/2 so that an OS/2 machine can be hooked to a Linux network and run its software. Also, a Windows emulator is being developed for Linux so that Windows applications will run inside Linux even though Linux is not a genuine Windows platform. Finally, programmers in Allemagne have created a conio.h and conio.c implementation that mimics the conio.h and conio.c files in Borland and Watcom C so that PC-DOS programs port more easily to Linux. And there is a PC-DOS emulator for Linux too. These achievements all attempt to allow someone to run a program on a platform it was not written for without changing code, and their common goal is portability.

Sancho: Do you think the knights will be able to do such a thing as the gooey face at all?

Don Quixote: I don't know, but I hope so.

Sancho: What stops them?

Don Quixote: Just a little cooperation.

Graydon Ekdahl is president of Econometrics, Inc. located in Chapel Hill, North Carolina. Graydon enjoys creating database applications and is interested in data structures, algorithms, C++ and Java. He can be reached at gekdahl@ibm.net.

Archive Index Issue Table of Contents

Advanced search

The Yard Relational Database System

Fred Butzen

Issue #49, May 1998

Yard is an RDBMS package that is published by Yard GmbH of Cologne, Germany.

- Manufacturer: Yard GmbH
- E-mail: yard@yard.de
- URL: http://www.yard.de/
- Price: $490 US (5 user license)
- Reviewer: Fred Butzen

I am a relative newcomer to Linux, having used it for about three years. In that time, I have been impressed by the creativeness of the Linux community, and its dedication to Linux.

Having caught the fever myself, I find myself telling my acquaintances in business about Linux and what it can do for them. Some are willing to listen and will use Linux for some narrowly defined tasks, such as a firewall. Most, however, are reluctant to use Linux as a part of their core operation. This is due in part to a reluctance to entrust their businesses to a free operating system —“free” somehow implying cheap or inferior. But a large part of the reluctance, I've found, is due to a seeming lack of applications that run under Linux.

In particular, the fact that none of the “big three” relational database packages are available in Linux editions is a serious problem for many business people. As businesses come to depend upon relational databases for managing their day-to-day operation, having a robust relational database management system (RDBMS) available is crucial to selling an operating system to business people.

Fortunately, a solution to this problem exists if one is willing to consider software written somewhere other than northern California. Many companies in Canada and Europe now offer their products with Linux support, and some of those products offer much of the functionality of better known products at a fraction of the price. One such product is the Yard RDBMS.

What Is Yard?

Yard is an RDBMS package that is published by Yard GmbH of Cologne, Germany. The Yard package includes the following:

- SQL engine
- Utilities for monitoring and managing the engine
- ODBC drivers and libraries for Windows 3.1, Windows 95 and Windows NT
- JDBC driver
- ESQL-C library: with this package, you can write C programs that have SQL statements embedded within them.
- X Window System-based tool for interrogating the database: this tool uses a point-and-click method for interacting with a database. With it, a user who has no knowledge of SQL can build queries and interrogate the database.
- Tools for building a CGI interface to a database

Implementation

Yard is a fully featured RDBMS. Its SQL engine includes the following features:

- Full implementation of the SQL-2 standard
- Stored procedures, implemented using the stored-procedure language described in the draft SQL-3 standard of August 1994
- Triggers, implemented as described in the draft SQL-3 standard
- Advanced data types, including **VARCHAR**, **TEXT** and **BLOB** (binary large object)
- A rich set of functions, including mathematical functions, date functions, string-manipulation functions and user-customized functions
- Methods for enforcing domain integrity and referential integrity
- Support for locale-specific data, including the use of national character sets for sorting, display of date and time and display of money
- Physical and logical logging

Yard uses a standard client/server interface: the SQL engine services requests from client processes that are running either locally or on networked systems. Local requests are received via pipes; network requests are received via sockets. Client processes can use a number of protocols to exchange data with the engine: ODBC, JDBC, ESQL-C, etc.

Resource Management

When you install Yard, you must define one or more *database systems* (DBS). Each DBS has its own shared-memory segment that is used by every user who is working with that DBS, and one or more *database spaces*, each of which is a chunk of disk that holds data. Yard's method of managing system resources closely resembles that of major commercial relational database packages; if you are familiar with Informix OnLine, you will feel at home working with Yard.

Each of Yard's database spaces is a statically allocated portion of disk. Usually, a database space is a part of a raw disk partition—one that does not have a file system on it. A database space can also be a file, although this is discouraged.

The system administrator must use a fairly complex formula to compute just how much disk space to allocate to a given database space. This formula includes the number and size of the physical and logical logs, the number of users, the number of tables and the estimated extent of each table.

The administrator can dictate the database space used by individual tables within a database. By estimating the frequency with which individual tables will have to be accessed, the administrator can balance disk I/O across all of the devices that hold a given database and thus ensure transactions are processed as quickly as possible.

As you can see, Yard is a serious package. To see it at its best requires serious hardware—preferably a machine with lots of memory, cycles to burn and multiple SCSI disks. It also requires an administrator who knows databases, UNIX and hardware and who can devote a significant portion of her time to monitoring and managing the database.

Versions

To use Yard, you must purchase a license for the SQL engine, plus licenses for one or more ancillary systems (ODBC, ESQL-C, YARD-X or JDBC).

Yard costs a fraction of what you would pay Oracle or Informix for a package with similar functionality. An SQL license for five users costs 990DM (not including value-added tax)—or about $490 US. (The actual cost will depend upon the rate of exchange between the Deutschmark and the US dollar, which fluctuates.) The other tools are similarly priced.

Unlike Oracle, which utilizes “named users” (that is, only a defined set of individuals can use the package), Yard defines a “user” as someone who is interacting with the engine. Thus, a five-user license can actually serve more than five users—a fact that is particularly important if your users work only intermittently with the database.

A free personal edition of Yard is available for download from Yard's web site (http://www.yard.de/). This package supports a single user and limits the size of the database. An ODBC driver is also available for the private edition. If you are interested in Yard, the private edition is an excellent way to become familiar with it.

Documentation

Documentation comes in the form of HTML files. Some sub-systems also have PostScript versions available; in particular, these are available for the ESQL-C library.

Documentation is available in German and English. I do not speak German but I found the English documentation to be well-organized and complete and its English to be both correct and lucid. My only complaint is that the HTML version tends to put each sub-section into its own file, making it difficult to print a copy to read while you're away from your computer.

The documentation assumes that you know SQL and are thoroughly familiar with Linux. Again, Yard is not a package for beginners.

Installation

Installation of Yard is driven by a shell script and runs smoothly. The script requests a location for installing the binaries, requests license numbers and keys, then copies the bits appropriately. A complete installation of the engine, libraries, header files and configuration files takes less than 11 megabytes.

Installation, unfortunately, is hampered by minimal documentation. The only documentation you receive are two lines of instruction, printed on the CD-ROM case, telling you to mount the CD-ROM and invoke the installation script. Thereafter, you're on your own. If something goes wrong, your only recourse is to read the shell script and interpret what it was doing when it failed.

Thanks to this flaw, installation is practically guaranteed to fail at some point, at least on your first try. For example, nowhere does the Yard package indicate that the binaries must be owned by a user and group named **yard**—your first hint of this requirement is the obscure error message you see when the script fails.

This may be splitting hairs, but it is a pain to diagnose and fix problems that could easily have been avoided had the publisher included a page of instructions in a README file.

Conclusion

Yard is a fully implemented, enterprise-scale database-management package. With it, you can process transactions for a small- to medium-sized business or not-for-profit enterprise. It offers most of the features of Oracle or Informix, but at a fraction of the cost.

Since it is an enterprise-scale RDBMS, Yard may be too much database for some users. If you are looking for a tool with which you can learn SQL or if you wish to set up a small database for your church or fantasy-baseball league, you probably would be better off with a more modest commercial package, such as JustLogic, or with one of the free databases, such as PostgreSQL or msql.

However, if you are a contractor who specializes in Linux-based solutions or a business person who is considering using Linux as the backbone of your enterprise's information system, you will find that Yard is serious software worth serious consideration.

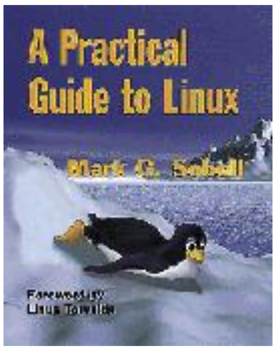
Fred Butzen is a technical writer and programmer who lives in Chicago. He is principal author of the manual for the Coherent Operating System, and is co-author of *The Linux Database* (MIS:Press, 1997) and *The Linux Network* (MIS:Press, 1998). He can be reached via e-mail at fred@lepanto.com.

A Practical Guide to Linux

Todd Sundsted

Issue #49, May 1998

This book is written for the “next” generation of Linux users—programmers, web designers and technically oriented people who are looking for an alternative to Microsoft's operating system and products—rather than the “hacker” generation (who brought Linux to this point).



- Author: Mark Sobell
- Publisher: Addison Wesley Longman
- E-mail: info@awl.com
- URL: http://www.awl.com/
- Price: $38 US
- ISBN: 0-201-89549-8
- Reviewer: Todd Sundsted

Mark Sobell's *A Practical Guide to Linux* is one of a growing number of books on the Linux operating system. It combines about equal parts topical guide and command reference. It is written for the “next” generation of Linux users—programmers, web designers and technically oriented people who are looking for an alternative to Microsoft's operating system and products—rather than the “hacker” generation (who brought Linux to this point).

The Linux universe is expanding at an ever increasing rate. Consequently, books of this type are faced with the challenge of being simultaneously comprehensive, relevant and up to date. To this book we must add a fourth challenge—as the title suggests, the material in the book must be practical. Of the three (or four), being comprehensive is perhaps the easiest to achieve.

Let's look briefly at what Mark Sobell offers us. Part I is organized into chapters, each covering a separate topic.

After a brief introduction to Linux and its features in chapter one, Mark immediately gets down to the business of using Linux in chapter two. In it, he teaches the reader how to log in, edit files, access the on-line manual pages and use the command line. It is pretty basic stuff that is necessary for the complete beginner.

In chapter three, Mark teaches the reader how to use the most common command-line utilities—tools like **cp**, **mv** and **grep**. Chapter three ends with a useful section on locating other users, communicating with them, and sending electronic mail.

Chapter four introduces the Linux file system. It describes the tree-like organization of the Linux file system, introduces files and directories, and describes how to work with them.

Chapter five introduces the command-line shell and related topics including input and output redirection, pipes and the file name wild card characters “?” and “*”. The material in this chapter is not specific to any of the common command-line shells, but instead introduces the features common to all of them.

Most users expect computers to have a graphical user interface (GUI). Therefore, the sixth chapter introduces the X Window System—the most common windowing system available to Linux users. Chapter six contains an introduction to the X Window System and its user interface components—buttons, sliders, and kin—and the mouse. It also describes briefly two of the most popular window managers available for X—MWM (Motif Window Manager) and FVWM (Feeble Virtual Window Manager) and provides information on customizing each.

In the seventh chapter, Mark introduces two very important topics—networking and the Internet. He describes the different network types and the various network utilities typically found on a machine running Linux. He also explains how to access both Usenet newsgroups and the World Wide Web.

Chapters eight and nine introduce the ubiquitous vi text editor and the Emacs text editor, respectively. While not the WYSIWYG writing tools many new users of Linux expect, they are inarguably an essential part of the repertoire of programmers, system administrators, web developers and others. The chapter on the vi editor is quite complete. The chapter on Emacs contains just enough material to get you going, but nowhere near enough to make you a master of this complex but powerful tool. No manual entry for p Chapter eleven introduces the topic of shell programming (or writing shell scripts).

In chapter fourteen, Mark introduces the tools of the programmer's trade—the C compiler, **make** and the source code management utilities. This chapter is easy to read but the material is not really necessary. Readers with any programming experience at all will find it far too basic, and beginners won't find enough information to make them into even fledgling programmers.

The final chapter of part one, chapter fifteen, introduces system administration. In this chapter the reader is taught how to boot the Linux system, backup files, install software, and rebuild the kernel.

Part II is a command reference that is quite well done. Each entry in the reference describes the syntax of a command, summarizes its operation, describes its arguments and options, provides a few noteworthy comments and illustrates several examples of its use. While there is nothing here that couldn't be obtained from a careful reading of the man pages for each command, the format is easier to read and the examples are far more useful. Occasionally an entry omits some of the less used features of a command. In those cases you'll have to refer to the man pages for the command or to other documentation.

Four appendices round out the book—one on regular expressions, one on accessing the copious Linux documentation available on-line (appropriately titled “Help!”), one on software emulators and one on POSIX and POSIX compliance.

My overall impression of Mark Sobell's book was positive. The chapters on the various command line shells easily took top honors for best of the book. Like it or not, the command line is an integral part of using Linux, and familiarity with one of the available shells is necessary to fully utilize its power. Users new to the Linux world will undoubtedly be daunted by the flexibility offered by the even the simplest shells—especially if their previous experience was limited solely to the DOS shell. A good introduction, however, goes a long way toward making the process of learning painless, and once learned, the user will find the flexibility and power exciting.

The chapter on vi was very solid. I only use vi when I don't feel like waiting for Emacs to start—that turns out to be quite often when I'm performing system administration tasks. Consequently, I use vi a lot more than I ever thought I would. I have a feeling that vi is here to stay and that learning to use it effectively is best done early.

The chapter on networking was a mixed bag. The information on networks and networking was interesting, as was the overview of NFS and NIS. The coverage of common commands such as **rlogin**, **ftp** and **ping** was also very useful. On the other hand, I don't think anyone uses **archie** or **gopher** anymore. (It did dredge up nearly lost memories of a much smaller Internet, however.) In fact, I'd bet many people haven't even heard of them. The material on browsing the World Wide Web, while accurate, is already beginning to go out of date. The material in the book is based around what looks like Netscape 3.x and Netscape 4.0 is already out, with a completely new user interface. Omitted is any mention of Java or Javascript.

I liked the command reference in part II of the book. While not a replacement for the on-line manual pages, it was fun to flip through off-line. The on-line manual pages are great when you know what you're looking for, but they're not much fun to browse. Part II, on the other hand, made good reading while waiting for a compile to finish or a page to load into my browser.

The least useful chapter, in my opinion, was chapter fourteen—Programming Tools. The material presented seems too basic for an experienced programmer, yet too superficial for a beginner. But then, as I think more about it, one group does come to mind—those programmers who are proficient in C or C++ but who have gained all of that experience while working in an Integrated Development Environment (IDE) on another operating system. Compiling and building an application from a command-line environment would be a big change. Chapter fourteen would help them get started.

I found the sections on customizing FVWM and MWM, in chapter six, to be too brief. I'm also concerned that, given the existence of two mutually incompatible (from a configuration file perspective anyway) but common versions of FVWM (1.x, 2.x and 95), the section on FVWM configuration might cause more harm than good for beginners. Perhaps Mark could have mentioned configuration, explained what pieces of each window manager can be configured, pointed the reader in the direction of the manual page and moved on.

I also found the inclusion of material on the various emulators to be of little use. While I consider both Wine and Executor to be two of the most impressive products I have ever seen, given their current (sometimes extreme) limitations they are unlikely to be useful to any more than a small minority of Linux users.

WABI and iBCS may have a slightly broader appeal, at least to those who need to run legacy applications, but neither emulator will replace the need for good native Linux implementations of solid application suites.

I would have liked a chapter on Perl. Like the command-line shells, Perl is a tool many users—especially those administering their own systems—will find useful. Indeed, whether one writes system administration programs, backup tools or CGI scripts, Perl seems to be the language of choice for a large number of experienced Linux users.

So, how did *A Practical Guide to Linux* do against the four challenges I mentioned earlier? It is definitely comprehensive (but that's the easy part). It's also relevant—most of the tools and utilities covered within its pages are here to stay. Aside from the material on the Internet (which admittedly is moving at a lightning pace), it is up-to-date. And, except for the chapter on programming, it is very practical.

The fact is, I'd buy it. It's every bit as good as any of the other Linux books available and better than many.

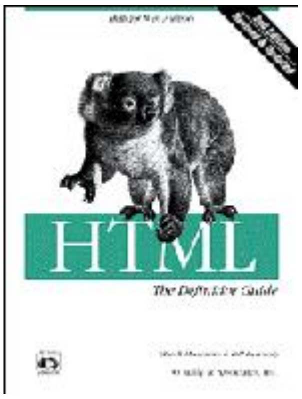
Todd Sundsted is a programmer, writer, and die-hard Linux enthusiast. He writes the “How-To Java” column for JavaWorld (http://www.javaworld.com/) and provides training and consulting through Etcee (http://www.etcee.com/). He can be reached via e-mail at tesundst@emss.com.

HTML: The Definitive Guide, Second Edition

Eric S. Raymond

Issue #49, May 1998

What is really outstanding about this book is the careful attention to HTML portability issues.



- Authors: Chuck Musciano and Bill Kennedy
- Publisher: O'Reilly and Associates
- E-mail: info@oreilly.com
- URL: http://www.oreilly.com/
- Price: $32.95 US
- ISBN: 1-56592-235-2
- Reviewer: Eric S. Raymond

Given the number of HTML books available, it takes something close to hubris to title a book *HTML: The Definitive Guide*. When O'Reilly sent me the manuscript of the first edition for review over a year ago, I was skeptical—but that first edition earned its title by presenting the best reference material I have ever seen on HTML. This second edition is a worthy follow-up.

The authors methodically walk you through every HTML feature in HTML 3.2, Netscape's extensions and Internet Explorer's extensions. They even cover such recondite topics as cascading style sheets. A handy reference appendix lists all the world's tags.

What is really outstanding about this book is the careful attention to HTML portability issues. Browser-specific tags and tag attributes are prominently marked. Charts like the summary of content-based tags on page 73, which tell you exactly how the tags will render under Netscape, Internet Explorer and Lynx, are alone worth the price of the book. And while non-portable constructions are carefully documented, the book is full of good advice about making your pages browser-independent.

Not only is this a definitive guide, it may be the only HTML book you'll ever need—at least, until the authors put out the next edition covering HTML 4.0.

Eric S. Raymond is a semi-regular *LJ* contributor who thinks Perl is pretty neat even though he still carries a torch for Scheme. You can find more of his writings, including his paper for the San Jose Perl conference, at http://www.ccil.org/~esr/. Eric can be reached at esr@thyrsus.com.

Protecting Your Site with Access Controls

Reuven M. Lerner

Issue #49, May 1998

Portions of your web site can be kept secure using user name, password combinations.

One of the wonderful things about the Web is that so much information is freely available. For the cost of a telephone call and a monthly bill from your Internet service provider, you can read hundreds of newspapers, get updates on the computer industry and listen to radio stations from your home town.

Even the most open, freely available site usually contains one or more sections that are not meant for public consumption. The reasons for cordoning off sections of the site can vary: Perhaps the webmaster wants a place to put his favorite hacks, a repository for testing new programs or a directory in which staff notices can be placed. If a site wants to charge for content or restrict access to members of an organization, the problem becomes even more obvious.

One popular way to handle these problems is to create a directory that others are unlikely to guess. But this approach, known as “security through obscurity”, only works as long as no one leaks the name of the hidden directory. A far more robust approach will restrict access based on user name,password combinations.

This month, we will look at ways in which to restrict access to your server with the Web's standard user name, password authorization scheme. The principles should apply to any web server, but I will be using the freely available Apache web server (available at http://www.apache.org/) in my examples.

How Access Restrictions Work

Access restrictions are part of HTTP, the protocol used in most web transactions. When your browser requests a document from a server using

HTTP, it is usually returned immediately, preceded by several headers (i.e., name,value pairs) describing its length, the date on which it was last modified and the type of content it contains.

HTTP's designers recognized that webmasters might want to restrict access to one or more directories. Since version 1.0, HTTP has included provisions for restricting access to parts of a web site.

Let's see how this protection works from a computer's view, first by looking at an unprotected site and then by looking at a protected one. Once we understand how access protection works, we can incorporate it into our own work.

Everything starts when a user asks the browser to retrieve a document. No matter whether the user types the URL into a text field, selects it from a list of book marks or clicks on a hyperlink in an existing page of HTML, the effect is the same. The browser takes the URL, dissects it into a protocol, a server and a document, and takes the appropriate action. In the case of a URL such as:

```
http://www.ssc.com/lj/
```

the protocol name is http, the server name is www.ssc.com, and the document name (really a directory) is /lj/. Most Web servers are configured such that requesting a directory is the same as requesting the file index.html within that directory, so the above URL is effectively equivalent to this one:

```
http://www.ssc.com/lj/index.html
```

We can simulate the browser's actions by dissecting the URL on our own and by requesting the document /lj/ from www.ssc.com using HTTP from the Linux command line. The TELNET program is generally used to log into a remote machine, most often to open a shell on that machine. By giving **telnet** an argument in addition to the machine name, we can specify the port to which we wish to connect. Since web servers sit on port 80 by default, we can connect to the web server on www.ssc.com by typing:

```
telnet www.ssc.com 80
```

When we establish a connection to that web server, we can enter an HTTP request. These requests start with a line describing the action we wish to take (known as a “method”), the name of the document we wish to retrieve and the version of HTTP we are using. Beginning with HTTP 1.0, this initial line can be followed by one or more header lines containing information about the user's browser, document types that the browser is willing to expect, HTTP cookies that may have been set in the past and other useful bits of information. For our purposes, it is enough to enter this line:

```
GET /lj/ HTTP/1.0
```

and then press **enter** twice—once to end the line containing the request, and a second time to indicate that we have finished sending all of the headers and that we will now wait for a response from the server.

If all goes well, the server will respond by returning a page of HTML. In this particular case, we will receive HTML-formatted text (as we can tell from the text/html Content-Type header at the top of the response) with the latest information about this very magazine. Your browser is responsible for taking the HTML returned by the server and displaying it for you.

Retrieving a Protected Document

If we try to retrieve a protected document, things get a bit more complicated. (We will see how to protect documents in just a moment; for now, assume that it is possible to restrict access to documents on a web server.) My main workstation, running Red Hat Linux 4.2 and Apache 1.2.4, contains a “private” directory whose contents are restricted. Let's retrieve the contents of /private/, just as I requested the contents of /lj/ before.

From the shell prompt, I connect to the web server with the following:

```
telnet localhost 80
```

Once I am connected, I request the “private” directory:

```
GET /private/ HTTP/1.0
```

Instead of receiving the contents of the /private directory or the index.html file contained within /private, I get the following response:

```
HTTP/1.1 401 Authorization Required
Date: Mon, 26 Jan 1998 12:08:17 GMT
Server: Apache/1.2.4
WWW-Authenticate: Basic realm="TestRealmName"
Connection: close
Content-Type: text/html
<HTML><HEAD>
<TITLE>401 Authorization Required</TITLE>
</HEAD><BODY>
<H1>Authorization Required</H1>
This server could not verify that you
are authorized to access the document you
requested. Either you supplied the wrong
credentials (e.g., bad password), or your
browser doesn't understand how to supply
the credentials required.<P>
</BODY></HTML>
Connection closed by foreign host.
```

In other words, my request was rejected because I had not authenticated myself. When did it give me a chance to do so?

Herein lies the dirty little secret of user authentication: when you retrieve a protected document, your browser really has to request the document *twice*. The first time that it tries to retrieve the document, a browser receives a message similar to the one that we received above, marked with the response code 401 indicating that you need authorization in order to retrieve this document.

Old or broken browsers stop at that point, presenting the server's error message to the user. Modern browsers that understand authentication present the user with a dialog box into which the user can type a user name and password. The browser then takes the user name and password, puts both into Base64 format and sends that along in an “Authorization” header after the initial request.

Modern browsers also save time by keeping track of user names and passwords that you have already entered. Thus, the first time you encounter a protected directory, you are prompted for your user name and password. The second time you retrieve a file from the same directory, you will not be prompted. Whether the browser waits to receive the **401 -- Authorization Required** error before sending the user name, password pair or it automatically responds to the message depends on the implementation.

Thus, if my user name is “reuven” and my password is “password”, I can retrieve the contents of the /private/ directory by using TELNET to access port 80 on my local computer and entering:

```
GET /private/ HTTP/1.0
Authorization: Basic cmV1dmVuOnJldXZlbg==
```

The first line is identical to what we have seen before; it indicates that we want to use HTTP 1.0 to retrieve the document named /private/ (which happens to be a directory, although the client does not know that) using the GET method. Rather than pressing **enter** twice after the first line, we only press it once and then add a single additional header. This one begins with “Authorization:”, meaning that we are about to send authorization information to the system using the “Basic” algorithm, which is nothing more than a Base64 encoding of the user name and password that the user entered in the form **username:password**.

If the user name, password combination succeeds, the system returns the contents of the resource requested by the browser. If the request fails, the same message (with response code 401) is returned to the user's browser. The browser can allow the user to try again or can display the error message sent along with the 401 message.

In this case, the user name, password combination does indeed work, giving me the contents of /private/, which is the file /private/index.html, returned in the following manner:

```
HTTP/1.1 200 OK
Date: Mon, 26 Jan 1998 12:41:14 GMT
Server: Apache/1.2.4"
Last-Modified: Mon, 26 Jan 1998 10:49:49 GMT
ETag: "1057-ca-34cc6a4d"
Content-Length: 202
Accept-Ranges: bytes
Connection: close
Content-Type: text/html
<HTML>
<Head>
<Title>My private site</Title>
</Head>
<Body>
<H1>My private site</H1>
<P>This is my private site.
From here, you can get to
<a href="test.html">my test page</a>.</P>
</Body>
</HTML>
```

The 200 status code at the top of the response indicates that everything has gone well and that the server was able to retrieve the document that we requested. As you can see from the Content-Type header (or simply by looking at the document's contents), the requested document contains HTML-formatted text. Were we to view this through our browser, we would undoubtedly see the text in different sizes.

Is This Real Security?

If you are wondering how I managed to get the Base64 equivalent of my user name, password combination, it was with the help of the following one-line Perl program:

```
perl -e 'use MIME::Base64;\
  print encode_base64("reuven:password");'
```

Entering the above in the shell results in:

```
cmV1dmVuOnBhc3N3b3Jk
```

which must have been the Base64 equivalent of **reuven:password**, because it allowed us access to the resource.

MIME::Base64 is a Perl module that you can get from CPAN (http://www.perl.com/CPAN/) for handling MIME-standard mail. I cannot remember the last time that I had to write a program to handle e-mail encoded with MIME, but the Base64 module comes in handy for non-mail applications such as this one.

If you have any experience with securing computer networks, you might be surprised to learn that user names and passwords are passed between web browsers and servers unencrypted. Indeed, while the text isn't passed completely in the clear, it would require another one-line Perl program to turn the Base64-encoded user name, password string back into its ASCII original.

Suffice it to say that this is not a very secure scheme. Someone monitoring packets sent over the network would have to work a bit harder in order to capture your user name and password, but not significantly harder than if the text were sent without any transformation.

At the very least, make sure to use user names and passwords that have nothing to do with /etc/passwd, the file that typically stores user information on Linux systems. Your secret documents can still be available via the Web, but your machine will not be open to break-ins which are a much more serious threat. (Someone who breaks into your computer can do much more than just read your documents.)

An authentication scheme known as “Digest” will soon be available. It is already available in Apache and is waiting for a browser to implement it. The digest method applies a function to a number of parameters, including the user name and password that are going to be sent, and a number generated by the server that is sent as part of the headers in the **401 -- Authorization required** response. The result of the digest function is then sent over the network, rather than the user name and password themselves. This is not a foolproof system, but it is far better than the current situation in which your passwords are easily available.

Creating a Password File

Now that we have discussed the theory behind all of this, we will take a look at what is necessary to protect directories on your server.

The first thing you need is a file in which user names and passwords can be stored. Apache comes with a program, **htpasswd** which can be used to create and modify such files. The syntax is fairly simple:

```
htpasswd [-c] passwordfile username
```

To create a new password file (or overwrite an existing one), use the following syntax:

```
htpasswd -c /etc/httpd/conf/passwords reuven
```

If you enter the above line at the Linux shell (with htpasswd in your **$PATH** environment variable), you will be prompted for a password. After you have entered the password twice, the user name, password pair will be stored in the file you specified.

The **-c** option creates a new file or overwrites an existing one. (This option is unnecessary to create a user; you can do that without the **-c** option, as described below.) Be especially careful with the **-c** option, because it overwrites old versions of the password file without warning or making backups.

To add a user to an existing password file or to change the password of an existing user, invoke htpasswd without the **-c** option:

```
htpasswd /etc/httpd/conf/passwords reuven
```

Regardless of whether you are adding a new user or changing an existing user's password, you will be asked to enter the user's password twice. When you have done that, the named file will be updated.

The password file contains nothing more than names and encrypted passwords in the format:

```
username1:password1
username2:password2
username3:password3
```

For example, the password file that I created for this column contains the following entries:

```
reuven:zZDDIZ0NOlPzw
reena:SjCCCbsjjz2Z2
foobar:RpubVfdhWwv1U
```

If you expect to handle many authorized user requests on your system and if the number of users on your server is high, you might want to consider using authorization using a more efficient system, such as DBM or DB. Support for DB and DBM are available for modern versions of Apache (although the appropriate module must be compiled in), as is support for a number of relational databases, including Msql and MySQL. More information on these options is available on the Apache web site.

Protecting Directories

Now that we have a list of user names and passwords in the correct format, we can use that list to protect the directories on our server. Each directory can use a different file containing user names and passwords—so your “top-secret” directory can have a different list of users than your “secret” directory.

There are two ways to protect files on your system. One is to put a file, called .htaccess by default, in the directory you wish to protect. This gives you the flexibility to modify individual directories quickly and easily and to give responsibility for different directories to the people in charge of those directories—but it also removes a certain element of central control.

We will thus look at the method in which access restrictions are defined in srm.conf, one of the Apache configuration files. Placing the access restrictions in srm.conf means you will have centralized control of access to your server, and you will have to restart the server each time you make changes.

Protected directories are declared in srm.conf within **<Directory>** and **</Directory>** statements with a relatively straightforward syntax. For instance, I added the following lines in this file to protect directories used in this article:

```
<Directory /home/httpd/html/private>
  AuthType Basic
  AuthName TestRealmName
  AuthUserFile /tmp/authusers
  require valid-user
  </Directory>
```

The first and last lines confine these declarations to /home/httpd/html/private, the protected directory on my server. Someone requesting a file within /home/httpd/html (the root directory on my web server) can do so without having to enter a user name or password. Someone trying to retrieve a file in /home/httpd/html/private (known as /private to the outside world), or in any subdirectory of /private, will have to enter a user name and password.

The user name, password pair is be passed using the “basic” authentication scheme that we saw earlier, in which user name, password is encoded using Base64 and sent as part of the HTTP headers following the request. Until browsers begin to support the “digest” method (or even more secure methods), all protected directories should declare the AuthType to be “Basic”.

AuthName is a way of identifying this directory to the outside world. You might want to call the directory something meaningful, such as “Joe's private directory”, or “FYI”. You might use AuthName to distinguish between different protected sections of your web server, such as “private area” and “staff area”. AuthName is generally displayed in the dialog box into which a user can enter her user name and password.

Next, we indicate which password file should be used for this directory. As mentioned earlier, each directory can use a separate password file, so it is important to specify which one you wish to use. If you expect to use more than a few password files on your system, you might want to investigate the use of groups, which allow you to grant privileges to different subsets of users in a single password file. (Users can be placed in groups, which we will not address here, but which allow you to associate each user in the password file with one or more groups).

Finally, we indicate that we will allow only valid users, meaning only those whose user names and passwords are in the password file named in AuthUserFile. You could also specify individual users who would be allowed into the site, such as:

```
require user reuven reena
```

Once you have placed this information in your server's srm.conf file, you need to tell the server to reread its configuration file. You can do this by shutting the server down and then restarting it or by sending it a HUP signal, as follows:

```
killall -v -1 httpd
```

This command sends a HUP signal (aka signal #1) to all instances of **httpd** currently running. Remember that Apache normally runs a number of servers simultaneously, so trying to identify individual processes and use the standard **kill** command is probably not a good way to go about it.

Once you have restarted the server, protected directories are only accessible to someone whose user name and password appears in one of these directories. If you want to test the protection mechanism, using TELNET (as described above) to pretend to be a web browser might be the best way to do it, in order to avoid a browser's cache of passwords.

Using This Information in CGI Programs

Just as you can protect directories containing HTML files and pictures, you can also protect directories containing CGI programs. For instance, if you want to make a selected number of CGI programs accessible only to a select number of users, you can define /cgi-bin/private in the same way as you did /private.

Here, for example, is the definition that I added to srm.conf in order to protect /cgi-bin/private:

```
<Directory /home/httpd/cgi-bin/private>
AuthType Basic
AuthName TestRealmName
AuthUserFile /tmp/authusers
require valid-user
</Directory>
```

As you can see, the definition is identical to that for /private, except for the name of the directory.

In this case, we will be asked for a user name, password combination if we try to execute a CGI program in this directory, using either GET or POST. (Apache allows you to set a separate access privilege for each method, so you could allow all users to GET but a restricted group to POST and still others to PUT and DELETE.) Before the request will actually be sent to the CGI program in question, we will have to authenticate ourselves.

One of the nice benefits of protecting CGI directories is that all programs in that directory immediately have access to a new environment variable, REMOTE_USER, which contains the name of the user in question. This is available to CGI programs written in Perl and using CGI.pm via the **remote_user** method, but all programs can retrieve the value of the environment variable.

How can this be of use? Well, we know that the user name must be unique; no two users can share a user name. Thus, we can use the user name as a primary key (i.e., a unique index) into a table in a relational database containing more information about the user—his or her age, interests and last visit.

Indeed, over the last few months, this column has looked at a variety of techniques for keeping track of information about users, most often by setting an HTTP cookie on the user's computer and setting a primary key value in the cookie.

The advantage of this system is that the user must verify his or her identity before being allowed to access the program—meaning that by the time the CGI program is executed, we can be sure that the user name exists, is associated with a real user and that this user represents that person (or has access to the user's password). HTTP cookies operate on a per-computer basis; if someone were to use my computer while I am not looking, they could retrieve information from all of the private sites from which I have retrieved cookies.

Another advantage of using this form of identification rather than cookies is that it gives the user mobility. No longer is the user tied to a particular computer or browser. While users must sign in before being allowed to use the site, they can access the site from anywhere rather than just from their computer at work or home.

There are disadvantages, too—the main one is the inherent insecurity associated with the basic authentication scheme. And some users prefer not to be bothered with having to enter their user name and password each time they visit a site. Such users would rather the site recognize and remember their settings automatically.

Listing 1 is a short CGI program written in Perl that identifies the user name entered. If this program is placed in an unprotected directory, it will indicate that no value for REMOTE_USER is available. If run from within a protected directory, however, it will return the user name that was used to access that directory.

If you were to create a table in a relational database (such as MySQL), you could define the primary key to be a user name of no more than eight characters. The value of remote_user could then be used as a reliable index into the database.

Protecting web sites is sure to be an increasingly important topic as the Web continues to mature. Apache is remarkably flexible when it comes to such security mechanisms. While I mentioned groups, there was not enough space to discuss additional options, such as restricting access by domain or IP address. See the Apache documentation for more information on this issue and the sidebar for additional sources.

While user name, password combinations are useful for restricting access to a web site, they can also be used to produce a unique key into a database. If you are thinking of creating a database to keep track of your users, you might want to consider using access controls to force users to log in.

Restricting access to directories on your web site is neither complicated nor difficult and lets you put sensitive or private materials on the Web without having to worry about someone discovering a secret URL.

Resources

Reuven M. Lerner is an Internet and Web consultant living in Haifa, Israel, who has been using the Web since early 1993. In his spare time, he cooks, reads and volunteers with educational projects in his community. You can reach him at reuven@netvision.net.il.

Archive Index Issue Table of Contents

Advanced search

Letters to the Editor

Various

Issue #49, May 1998

Readers sound off.

Japanese Word Processor

I have been a subscriber to your fine magazine for several months now, and each month I look forward to receiving the newest edition in the mail. Each edition is better than the last, and you folks always seem to cover issues I need more information about right around the same time the magazine arrives (are you psychic?). Perhaps the following might be of interest to your readers.

Would you like to work on an exciting project? There is a Windows application, called JWP—a Japanese Word Processor. This package was written by Stephen Chung, and as a GNU product it is freely distributable. I've used it extensively over the past few years, and it is a *great* package.

This project will never get off the ground without volunteers; therefore, I invite any interested X-Windows developer who wants to make a contribution both to the GNU and Japanese-speaking communities, to lend a hand with this exciting project.

The JWP-Port Project home page contains more information on the JWP package as well as the JWP-Port project itself. If you are interested, please visit the page at http://qlink.queensu.ca/~3srf/jwp-port/.

—Steve Frampton 3srf@qlink.queensu.ca

ispell

I just read your *LJ* article on **ispell**. ["Take Command", February 1998] You obviously like it. I find it a large pain in the ass, and wish I had a normal UNIX spell-checker available on my Linux box.

I have two gripes concerning ispell. First, the word list it comes with is not very complete. I've added 382 words so far and keep adding. One reason for needing to add so many is that ispell (for reasons I have argued with its creator about) insists on trimming “'s” from possessives. That means that sooner or later I find myself having to add the possessive form of every noun in the language to my word list. And that's my second gripe: why can't they at least provide the removal of “'s” as an option?

The developer of ispell insists that this bug is a feature. I think his feature is a bug. What do you think? Doesn't this bug you?

—Andrew T. Young aty@mintaka.sdsu.edu

In general, ispell works well for me, although I do have my own frustrations. I'd like it to have a “change all” feature, so if the word is consistently misspelled throughout I only have to tell it to change it once. There are words I think should be in there that aren't: hydroponic and oxymoron, for example. I've added 304 words but many have been proper names (people and products), computer jargon and abbreviations.

The “'s” is a feature in the spell checker I have at home too; at least for words I am adding to the dictionary. I too find it annoying, but not as much as you evidently do.

—Editor

Linux

I just read the Letter to the Editor from searoy@aol.com [February 1998] and thought I needed to share with you our experience with Linux at work. First of all, you do not have to become a programmer to run Linux and many of its applications (my two sharpest guys are operators). The Red Hat 5.0 distribution (it costs a whopping $39 US) is very easy to install on Intel computers and even runs on Sun SPARCs and DEC Alphas (you need to be familiar with how SPARCs and Alphas work through the boot process). Red Hat comes with editors, databases, compilers, scripting languages (Perl, Expect, etc.), network security tool and several types of servers (web, SAMBA, printer, Novell, etc). Even the updates are easy to get off of the web and install on the computer.

The second thing we like about Linux is that it is supported by programmers, hackers, engineers and users around the world, so problems get fixed fast. I recently read an article in BYTE magazine about the bug in the Pentium processors, and the only OS to have a fix out was Linux. (The bug is in a piece of code that will try to stuff a 64 bit word into a 32 bit register.)

The third thing we like is that we learn how the computer, operating system and applications work, which has a snowballing effect. Each time we've solved one problem or gotten a program working it has helped on the Solaris and HP-UX systems we have. Last, you can get into Linux cheaply. Find an old 386/486 with 16MB of memory and at least 300 MB of disk space and you have a computer system for experimentation. Kevin (one of my operators) built a firewall for the local college with a 386/33 that they had in the scrap pile, and we run all of our network monitoring tools on 486/66 ASTs that were headed for salvage.

—Bernie Morin bm@aol.com

“A Partner's Survival Guide”

Brilliant! Absolutely loved it—“The two are merely coincidental.” This glimpse into a hacker's life is a big, unattended part of the mainstreaming of Linux. [“A Partner's Survival Guide”, Telsa Gwynne, February 1998]

—Arnim Littek arnim@digitech.co.nz

September 1997 Issue

The article by Andy Vaught, “Introduction to Named Pipes” [September 1997] contains an error. Near the bottom of the first column on page 32 is the following command:

```
   mkfifo pipe;  ls -l pipe1;  cat < pipe
```

The above is worthless, as **pipe** should in fact be **pipe1**. This error caused *me* no trouble, but it was not intended for me, rather for someone who has paid money for *LJ* and expects to learn and trusts *LJ* to be accurate. Now, “No finger pointing between *LJ* and Vaught”. I would just like to see the guilty party stand up and apologize to ALL the readers for the frustration they have caused someone trying to learn. As a publisher, you have an obligation to ensure that there are no errors. And, please, no excuses.

—August Gramm asursa@cris.com

You are correct: the commands should have read ***pipe1*** *in all three cases. You are also correct that LJ* has an obligation to publish technically correct information. We are, however, not perfect. We would like to be, and we have at least four different people look at each article. We are continually surprised when mistakes like the one you mention get past us. The ultimate responsibility for mistakes lies with me—my policy has always been “the buck stops here”. I am sorry for any frustration caused by these typos. Yours was the only letter I received about this error and the September issue is quite old—perhaps even the newbies were able to figure out the right way to give the commands.

—Editor

Oracle In-house Linux Port? Bah!

Your piece on databases for Linux (“Databases”, February 1998) mentions a stealth, “in-house” port of Oracle for Linux, which we've apparently had “some time.” And Oracle refuses “to sell or support it”.

Where do these rumors get started? I've worked for Oracle for two years, and have been a Linux-head that whole time. If a version of Oracle written for Linux existed, I'd have noticed.

Oracle7 for SCO certainly does run on Linux under iBCS, and quite well. Maybe a misunderstanding of that fact somehow started the rumor.

No one here at Oracle has ever heard of an actual Linux port. If one does exist, perhaps it's being used to help reverse-engineer those captured UFOs at Area 51. That might explain the super secrecy.

—Steve Abatangle, Oracle Corporation sabat@us.oracle.com

At the 1994 Uniforum Conference, a man wearing an Oracle badge walked up to the Linux Journal booth and introduced himself as an Oracle developer to our publisher Phil Hughes. This man told Phil he had a working version of Oracle on Linux. Unfortunately, Phil has forgotten the man's name, though not the event.

—Editor

Bleeding Edge Articles?

I am a long-time reader of *LJ* and have always been very pleased by your articles. Linux is finally getting some of the respect it so richly deserves, and it is great to see such a fine publication supporting the cause.

I do have a question, however. I'm an occasional kernel developer and long-time Linux user and I would like to see more articles (maybe one per issue) on Bleeding Edge Linux projects and ports. Articles detailing such relevant topics as MacLinux (Linux for Macintosh/m68k, http://maclinux.wwaves.com/), Linux/PMac, GGI, and other works in progress would definitely be a boon to your readers and allow for more people to become involved in these experimental projects. Now, in the true Linux fashion, I'm not going to suggest you do things without volunteering myself in the process and I would like to know if you accept articles from the user world and (if so) to whom can I send them?

—Joe Pranevich knight@wave.lm.com

We have had articles on the Macintosh and Linux in issue 31, issue 37 and issue 45. Reuven Lerner talks about CGI each month in “At the Forge”. I agree it would be nice to have bleeding edge articles each month, so we print them as we can. Yes, we do accept articles from the user world. Please send your ideas to info@linuxjournal.com. Author information can be found on our web site at http://www.linuxjournal.com/wanted.html.

—Editor

BTS

I think the first “corrupted terminal” answer on page 72 of the February 1998 issue needs some work:

1. **cat** was NOT “designed to handle ASCII files.” **cat** conCATenates files to standard output (hence the name). It concatenates one or more files from argv, or if argv is empty, simply from stdin until EOF. There is nothing in cat or its documentation to suggest that it was designed for ASCII files, or could not handle binary files.
2. **cat** does not “interpret a lot of the binary file as control sequences.” The terminal emulation supported by the console code (or an xterm) interprets any control sequences it is sent via any program that writes to it. These can come from any program. The terminal emulator is supposed to do this. It's why things like Pine and Emacs work, for example.

Here's a question for you: how many Linux seats are there, and how many new seats a month are there? Any idea?

—Ron Minnich rminnich@sarnoff.com

No idea. Vendors don't like to give their sales numbers to their competition, and those numbers would probably be the most accurate count. There is the Linux Counter, but not everyone knows about it or takes the time to register so its count is way low. If you haven't registered, go to http://counter.li.org/ and do so—now!

—Editor

Red Hat CDE

I recently saw the review of Red Hat CDE by Don Kuenz in the January edition of *LJ*, and decided to buy the software. However, on filling in the order, I saw that the January price for CDE was 25% higher than the price advertised in *LJ*. I think it's unfortunate that the information in the published review was out of date before I even received the magazine. I am also disappointed that Red Hat, while obviously benefitting from the PR of this review, are not prepared to honour the advertised price.

—Dr. Steven Bird Steven.Bird@edinburgh.ac.uk

Archive Index Issue Table of Contents

Advanced search

Open Source Debate

Phil Hughes

Issue #49, May 1998

As part of Eric's article, he introduced a new phrase for talking about software such as Netscape Navigator as well as other software where the source code is freely available. The phrase is Open Source.

Last month we published an article by Eric Raymond on Netscape's decision to release the source code for their web browser. A lot has happened since Eric's article, and that *lot* is what I wish to talk about this month.

As part of Eric's article, he introduced a new phrase for talking about software such as Netscape Navigator as well as other software where the source code is freely available. The phrase is *Open Source*.

The idea for a new name and the choice of that name has been an ongoing debate and, at this writing, continues. The confusion is associated with the term *Free Software*. Free has many different meanings. In particular, free can refer to price or to freedom. While freedom is the issue that we are attempting to address, there has continued to be confusion.

Rather than attempting to define what we mean by *free*, the introduction of a new term seemed in order. With the new term comes the opportunity to define it.

Bruce Perens (of Software in the Public Interest) and I decided that a web site with a repository of information on Open Source was the best approach. That web site is at http://www.opensource.org/ and includes the definition of Open Source and extensive resources to get others on the Open Source bandwagon.

Netscape Update

By the time you read this magazine, the source code for Netscape Communicator 5.0 Standard Edition should be available. Netscape announced they would release this code on March 31, 1998.

Netscape released a preliminary version of their license agreement (Netscape Public License) for public comment. That comment period ran through March 11.

Netscape put the proposed license on the Web along with a FAQ and annotations. The ongoing saga of this license is available on the Web at http://www.mozilla.org/. I'm not a lawyer but what I see of the license makes sense. It isn't a GPL and it isn't a BSD license, but that is because neither makes sense for Netscape.

One thing in the license I found interesting is they have elected to keep the name Netscape for branded products and allow the use of Mozilla for derived works. This makes sense to me. When I was talking to Eric Raymond before he met with Netscape, I brought up the concern that a bad port or modification of a Netscape product could reflect poorly on all Netscape products. The decision to use a different name offers the needed freedom, yet allows the consumer to differentiate between a Netscape product and a derived product.

Other Players

Netscape is not the only commercial vendor in the Open Source camp; there are others.

One hardware vendor who got into this camp before it even existed is Cyclades. Cyclades was the first communications board manufacturer to embrace Linux. They have always made the source code available for their drivers and their sales growth in the Linux and BSD markets has supported their decision to release source code for their drivers.

Another long-term vendor in this community is Cygnus Solutions. They offer commercial support for open source software and continue to grow and evolve with the community.

Of course, all the Linux vendors are there by default as well as the Apache web server and the Perl programming language. Vendors who elect to get involved with these products become part of the Open source vendor base.

Resources

I have already mentioned the Open Source and Mozilla web sites. As always, we put late-breaking Linux news up on the Linux Resources web site, http://www.linuxresources.com/.

1998 Open Systems Products Directory

Archive Index Issue Table of Contents

Advanced search

gprof, bprof and Time Profilers

Andy Vaught

Issue #49, May 1998

Mr. Vaught shows programmers a few commands to determine which sections of their code need optimization.

A friend of mine was explaining to me why he thought his program wasn't running fast enough. “Have you profiled it?” I asked. “No, but I'm pretty sure I know where the bottleneck is,” he replied—famous last words. “Well, let's try the profiler,” I said. Profiling quickly revealed that 98% of the CPU was being spent in one subroutine of my friend's program and that 86% of the CPU was being spent in *one line*, and he was wrong about the location of the bottleneck.

Profilers are invaluable tools that let you know where a program is spending most of its time. This information is extremely valuable because it tells you where your time can best be spent in making your program more efficient and where you are wasting your time. Typical programs are not quite as lopsided as in the above story. The “80-20” rule says that a program will spend 80% of its time in about 20% of the code.

If the total running time of a program is *n*, then we can break up the running time into pieces:

```
total-time = a1*n + a2*n + a3*n + ...
```

where the “a”s represent fractions of the total time that your program spends in a particular segment. The sum of the “a”s must add to one. The 80-20 rule says that one of the “a”s will be quite large. For example, suppose a1 is 8/10 and a2 is 2/10. These numbers correspond to a program which spends 80% of its time doing a1 and 20% for everything else.

```
total-time = .8*n + .2*n
```

Now suppose we optimize a2 so that it runs twice as fast as before—a significant speedup. The time 0.2*n is now 0.1*n. The total running time is now

0.8n+0.1n = 0.9n, meaning the whole program executes in 90% of the time that it originally did. Suppose we instead concentrate on the other piece. If we halve the running time of the first piece, it becomes 0.4n. 0.4n+0.2n=0.6n, or 60% of the original running time. As you can see, it is worth our while to concentrate on the particular portion of the program that dominates the runtime.

In my friend's case, we were able to optimize that single line a bit. The real optimization came two days later when he told me that he had removed the whole subroutine in question, simply by changing how he thought about his data.

The easiest profiler to use under Linux is the **gprof** profiler. **gprof** is a standard part of the GNU development tools. If you have **gcc** installed, you probably have gprof too. To use gprof, simply recompile your program with gcc using the **-pg** switch. This option causes gcc to insert a bit of extra code into the beginning of each subroutine in your program. The **-pg** switch must also be used when you link your program, since another snippet of code must be present to tie the pieces together.

After recompiling, run your program. It will execute slightly slower because of the work needed to profile the code, but it shouldn't be too slow. After the program finishes, there will be a file named gmon.out in the current directory. This file contains the profiling information collected during the program's run. **gprof** is used to print this in a human readable form.

gprof outputs information in two ways: a flat profile and a call graph. The flat profile tells you how much time the program spent in all of the subroutines, and the call graph tells you which subroutines called which subroutines.

The first part of a flat profile is shown in Listing 1. The “self seconds” column shows how much time was taken up by each subroutine. The number of times each subroutine was invoked is shown in the “calls” column. The “self ms/call” columns gives the average time in milliseconds spent in a given subroutine, while the “total ms/call” includes time spent in subroutines called by that subroutine as well. For reasons explained in the gprof documentation, this last column is actually only a guess and should not be relied upon.

In this example, an important thing to note is that the **mcount** subroutine is the actual profiling subroutine call inserted by gcc when it compiles code with the **-pg** switch. The fact that the program spends nearly 20% of its time here indicates that a lot of calling and returning is happening, and that one way to speed up the code would be to eliminate some of the subroutine calls. Which ones? The subroutines **rnd** and **uni** are likely candidates, since they are called 142 million times in 900 seconds.

The other profiler in common use on Linux is **bprof**. The major difference between bprof and gprof is that bprof gives timings on a source line basis while gprof has only subroutine-level resolution, and also includes information like invocation counts. To use bprof, link an object file, bprof.o, into your program. After you've run your program, a file named bmon.out contains the timing information. Run bprof on this data file, and it makes copies of your source files with timing numbers prepended to each line.

How the Profilers Work and Why They Sometimes Don't

The profilers under Linux work by examining the program counter at regular intervals to see where in the code the program is actually working. Although easy to implement, there is a randomness to this process that results in a certain amount of noise in the measurements. Over “long” intervals, the amount of good data overwhelms the noise. This is usually good enough in practice, when we are only interested in finding the bottlenecks.

Both gprof and bprof use timers that only run while your program is actually running. One of the ways that the kernel can use a lot of time on your program's behalf is by reading or writing huge amounts of data. These times do not accumulate into the profile. A more subtle way in which the kernel can eat a good deal of time is if your program uses so much memory that the kernel must swap part of your program in and out of memory. This situation is called page thrashing because you can usually hear your disk thrashing around.

A simple way of checking the system time is to run your program with this command:

```
time <
```

After your program finishes, three times are printed: user time (the time the CPU spent running your program), system time (the time the CPU spent in the kernel serving your program) and elapsed time (real time, sometimes referred to as “wall clock” time). By comparing these times you can get a rough view of how much work the kernel is having to do for you.

My favorite profiler is a program called **pixie**, which is unfortunately not available for Linux. Pixie works by actually reading the executable, inserting counting code into “basic blocks” of code that can only be entered at the start and exited at the end. Support exists in gcc today for counting execution of these basic blocks (the **-a** option), but getting actual times for each block is not yet supported.

Using the Profile

So now you know the location of the bottleneck in your program. There are a couple of simple techniques for making things go faster. The easiest is of course to use the **-O** flag of gcc to optimize the code. Be warned that optimizers are notorious for generating bad code.

It is often possible to decrease running time in exchange for an increase in space. Consider the following (FORTRAN) code fragment in Listing 2.

Profiling revealed that the program was spending over a quarter of its time in this loop, not just because it is slow, but because it was also being called frequently. Since it is called frequently, the variables cx, cy and cz are recalculated for each loop iteration. If we precalculate these values into the **tcentr()** array, four array references, three floating-point additions and a multiplication are replaced by a single array reference. A lot of code in a critical loop is thereby eliminated.

The cherry on this is moving the multiplication by 0.25 (multiplication by 0.25 is faster than dividing by 4.0) out of the loop altogether; thus, instead of multiplying each element of the sum by 0.25, we multiply the whole sum by 0.25. Since the loop happened to execute about 100,000 times or so, we've eliminated 99,999 floating-point multiplies. The new code is shown in Listing 3.

At this point, the program was profiled again, and this subroutine had dropped to taking 15% of the total time with 10% being taken up by the square root calculation. Since a different subroutine was now dominating the run time, the focus of the optimization effort moved away from this subroutine.

There is one more semi-easy thing that can be done for this code. It happens that the square roots are known to be needed for a very limited range of values. So, we can replace the square root function with a function that outputs the same value by looking up precalculated values in a table and returning an interpolated value for values that occur between table entries. Again, we have traded space for time.

Another common way of speeding up a program is to replace array references with pointers. In the example at the start of this article, the line that accounted for 98% of my friend's program looked like:

```
int i, array[1000000];
...
 i = 0;
 while(array[i] == 0) i++;
```

This line searches for the next non-zero element of an array. The operation of referencing the array consists of multiplying *i* times a scaling factor (which is implemented as binary left shifts), adding this value to the start of array[] and fetching from that location. Replacing this code with:

```
   int *p, i, array[1000000];
   ...
    p = array;
    while(*p == 0) p++;
    i = p - array;
```

eliminates the scaling and addition and sped things up by about 10%.

Another change we tried in this case was unrolling the loop. The code is replaced by:

```
   int *p, i, array[1000000];
   ...
    p = array;
    for(;;) {
    if (*p++ != 0) break;
    if (*p++ != 0) break;
    if (*p++ != 0) break;
    if (*p++ != 0) break;
    }
```

The idea here is that on certain types of machines, taking a branch is expensive while rejecting a branch is cheap. In a very tight loop, the overhead of the loop can end up being a significant part of the total time. The code in the second example has been rewritten so that most branches are not taken and that more work is done in the body of the loop for each iteration of the loop.

As it turned out, this “optimization” didn't speed anything up for the machine we were using. A good compiler compiling with **-O** will unroll short loops for you. It is important to profile before and after to see if what you've done has helped or hurt.

The other main option for optimizing code consists of simply looking for a better algorithm. For example, suppose we want to search an array for a particular entry. If the array is very small, we can simply check each element in turn. When the number of elements becomes large, hash tables are a quick and easy way to prune the number of elements that must be searched. For data that cannot be hashed, tree seaches provide another alternative. Hashing and trees are beyond the scope of this article, but should be a part of any programmer's bag of tricks. Any good book on data structures can show you how they work.

Machine specific optimizations are generally best left to the compiler. Compilers are becoming quite good with the simplest sort of optimizations, and gcc is one of the best. Once the profiler has located the slow portion of the program, the best way to optimize it is to simply imagine having to calculate everything by hand. Hopefully, you will notice improvements that a compiler will miss.

After all, the perfect compiler will never be written. There is an old joke that once a computer is built that can write code as well as a person, that computer will expect to be paid for its work. Hey, it's job security.

Andy Vaught is currently a Ph.D. candidate in computational physics at Arizona State University and has been running Linux since 1.1. He enjoys flying with the Civil Air Patrol as well as skiing. He can be reached at andy@maxwell.la.asu.edu.

Archive Index Issue Table of Contents

Advanced search

Linux on Track

Harald Kirsch

Issue #49, May 1998

Linux was used in two projects as a data acquisition system running more or less autonomously in the German ICE trains. This article describes design issues and implementation as well as the problems and solutions used in those projects.

The Fraunhofer Gesellschaft is a non-profit research organization specializing in applied research and acting as a proponent of technology transfer between basic research and industry. With nearly 50 institutes in Germany, almost all aspects of science are covered. Part of the IITB (Institut für Informations- und Datenverarbeitung) specializes in applications of computer-based monitoring, control and diagnosis of industrial processes and equipment by means of signal and image analysis. As such, our group was involved in two projects requiring data acquisition and analysis in one of the German high speed trains, the InterCity Express (ICE). This article describes how the data acquisition was implemented using Linux.

The Projects: UNRA and ICE-D

Project UNRA (unrunde Räder) tried to discover the reasons wheels of high speed trains become out-of-round sooner than expected. As railway experts know, train and coach wheels become out-of-round, i.e., the difference between the minimum and the maximum radius of a wheel becomes non-negligible. For ICE wheels, DB-AG uses 0.6 mm as the threshold above which wheels must be changed. Despite being only 0.1% of the wheel radius, such differences induce low-frequency vibrations into the coach structure. They not only put additional stress on structural materials but also cause an unpleasant ride for passengers.

ICE High Speed Train

As part of this project, a measuring axle developed and patented by Forschungs- und Technologiezentrum der DB-AG (Minden, Germany) was installed in a regular running ICE to measure triaxial forces in the stand-up point of the wheel. In addition, four accelerometers were installed, three on the axes at the bearing and one in the coach measuring vertical acceleration. In addition, a resolver was attached to the axle. It delivers 1440 TTL-edges for one turn of the wheel. The edges are used to clock 90 A/D conversion for one turn of the wheel, resulting in a sampling rate dependent on the wheel's rotation speed but synchronous with its rotation.

With additional channels not mentioned above, a total of 17 channels had to be measured. Since the radius of an ICE-wheel is 460 mm, 1 km amounts to:

[redacted]

samples, resulting in:

$$\frac{31139}{\mathrm{km}} \cdot 17\,\mathrm{channels} \cdot 2\,\frac{\mathrm{byte}}{\mathrm{channel}} \cdot 1500\,\mathrm{km} \approx 1514\,\mathrm{Mbyte}$$

of data every day.

The second project, called ICE-D, did not have such a great demand on bandwidth, but it required some data analysis to be computed on-line. Since the system is quite similar to the one used in the UNRA project, this project will not be described in such detail. Suffice it to say that another Linux computer will run for two years in a coach car with a new type of bogie (the assembly of four wheels on which the coach rests) to acquire and analyse 50 data channels.

Hardware

The computer hardware used in both projects is not identical but not very exciting in either case. In UNRA we use a garden variety Pentium 90 system with Adaptec 2940 SCSI controller, a sufficiently large Quantum disk, which seems to have settled its disagreements with the SCSI controller, and an HP DAT streamer. For the ICE-D project very similar hardware is used. Comparatively expensive were the 19-inch cases which were necessary to mount the computer in a rack in the train.

Of more interest might be the measurement hardware. In UNRA there is an Analog Devices RTI-860, which is a 16-channel, 12-bit A/D (analog/digital) conversion board. What makes it particularly suited for use with Linux is its on board memory of 256K samples, relieving us from hard real-time constraints.

Another board, called ADCO, was developed along the specifications of IITB by CMS. It is basically a device to measure times with a 40 MHz clock driving a 16-bit counter. On external events, the counter is read into a FIFO of one kilometer word length and then reset to start again from zero. Consequently the FIFO collects counter values representing a sequence:

$$\Delta t_1, \Delta t_2, \ldots$$

of times between events. The events are generated by the resolver mounted on the axle and happen once for every four degrees of wheel rotation. Knowing how much time the wheel needs to rotate four degrees, we can calculate the rotational speed of the wheel with high precision. The one kilometer sample FIFO on the ADCO is small compared to the buffer on the RTI-860 and actually proved to be too small for the first driver developed (see below).

Project ICE-D required another kind of measurement hardware because the task was not to acquire data at high speed but to measure up to 64 channels. We decided on an RTI-834 from Analog Devices because it can measure 32 channels and is probably the only board around with a useful programmer's manual. The manual is not free (approximately 250 DM) but, believe it or not, it contains almost all details about programming the hardware, including examples that made it comparatively easy to write a device driver.

In addition to these devices, a GPS (Global Positioning System) device was mounted on the coach. Position information was recorded with the data to be able to correlate it later with actual locations on the track.

Software

The software comprises several main components: the data acquisition program, a watchdog and a taper, a GPS monitor and device drivers. These components are described in the following subsections.

Data Acquisition

Except for the time between two and four o'clock in the morning, the data acquisition is active and digitizes data. Because the data acquisition is synchronous with wheel rotation, the data rate depends on the train's speed. At 300kph the wheel rotates at about 29Hz resulting in a data rate of

$$29\,\mathrm{Hz} \cdot 90 \cdot 17\,\mathrm{channels} \cdot 2\,\frac{\mathrm{byte}}{\mathrm{channel}} \approx 89\,\frac{\mathrm{kbyte}}{\mathrm{s}}$$

which has to be streamed to disk without loss.

Every 345 rotations of the wheel (about one kilometer), the hardware is reset to trigger again on the next zero-degree marker of the resolver. At that time the data acquisition program fetches the most recent information from the GPS, writes it to file, closes the file and opens a fresh one. Each file covers one kilometer of track and is nearly one megabyte in size. This approach was chosen for several reasons:

1. One megabyte and one kilometer are convenient sizes to handle with data analysis software.
2. Synchronising every kilometer makes sure that losing individual events from the resolver due to noise will not spoil all data for the rest of the day.
3. One kilometer was determined to be a useful checkpoint to record GPS information.
4. The files are not created anew each day but are overwritten for efficiency reasons. In case of a power failure, it is almost impossible to find out how much of a file is new and how much is from the day before; therefore, a partly written file has to be thrown away. Throwing away up to one kilometer of data is a reasonable tradeoff between number of files and amount of data lost.

Of course, there is nothing magic about one kilometer. Two kilometers or one half kilometer would probably have worked equally well.

While reading data from the devices, the data acquisition program also monitors the wheel's rotational speed to check whether the train's speed is above 60kph. Below that threshold, data is considered to be of no interest and is thrown away. In particular the file currently being written is reset and reused as soon as the speed rises above the threshold. Of course, up to one kilometer worth of data recorded at speeds above 60kph is discarded, but in fact, the threshold of 60kph is a rough guess anyway so no harm is done by discarding some data recorded at speeds slightly above 60kph. Typical travelling speeds of the ICE are, depending on track type, 100kph, 160kph, 250kph and 280kph, and only those speeds were of major interest in the project.

The data acquisition program is rather simple, most of it doing error handling in case of read or write errors. Since device drivers were implemented for the RTI-860 as well as for the ADCO, digitizing is as easy as opening a file and reading from it. The only thing requiring even minimal thought was that the data rate from the two drivers is not identical. Reading the same amount of data from both devices in every course through the main loop would soon fill up one of the driver's buffers. A general solution in such cases is the use of the **select()** system call; however, in the given case, the exact ratio between the two data rates was known and the amount of data read from each driver in every read-call was chosen accordingly.

cron Jobs

At two o'clock in the morning the data acquisition process stops recording data in order not to interfere with other work done at that time. First, a **cron** job reboots the system as a preventive measure against memory leaks. Although none were observed, rebooting costs nothing and does no harm. After the boot, the acquired data is written to tape with a script started as a cron job which ultimately calls tar.

A minor nuisance was that it is almost impossible to find out how much space is used on the tape if internal compression of the DAT drive is enabled. Assuming that the compression ratio is about the same every day, it would probably have been possible to put two days' worth or 1.5GB of uncompressed data onto a 2GB tape. Since the A/D converter only delivers 12 bits which are stored as 16-bit values, a compression to 1.125GB should be trivial. Another 12% reduction is probably possible because most of the time the digitized signals do not cover the full 12 bits.

During the rest of the day, i.e., not between two and four o'clock in the morning, another cron job is started every ten minutes. As a measure against yet unknown bugs in the data-acquisition program which may cause it to crash, a watchdog program checks if the data-acquisition process is still in the process table, and if it is, assumes that it is doing something useful. If it is not in the table, the system is rebooted. As of this writing the watchdog has still to prove its utility, since no such incident has been found in the log files.

GPS monitor

The GPS device is a cute little gadget that looks like a computer mouse without buttons. It is about the same size, shape and color as a mouse and is mounted on the top of the coach to have a clear view of the sky. It is connected to the computer via a serial line which also delivers power to the device. As soon as the GPS is connected, it starts sending several types of information which can be read with a command as simple as:

```
cat /dev/ttyS1
```

as long as /dev/ttyS1 is set to the correct baud rate. By writing to the device, it can be programmed to deliver only certain types of information.

The high speed data acquisition has one minor deficiency—it only delivers a dataset once per second. As described above, the positioning information is entered into every data file one kilometer in size. Now suppose that the data acquisition process starts reading from the GPS after it has acquired the last one kilometer sample. Reading may take up to one second while the wheel turns up to 28 times per second, thereby losing about 80 meters of data.

Since losing data was not considered efficient, the **gpsmonitor** program was introduced, running parallel to the data acquisition process. It reads the position information at the given rate of one second and stores it in a file where the most recent information is always available for the data acquisition process.

To make sure that the data acquisition process does not read partially written data sets, in general it would be necessary to use a file locking scheme to bar the acquisition process from reading while gpsmonitor is writing its data. However, one data set is only 80 characters in length and is sent to the file in one write-operation. Checking the Linux kernel sources might show that this is not an entirely atomic operation, but experiments with a process rereading the information at the highest possible frequency have shown that the probability that a write of 80 characters would be interrupted by another process is practically zero, i.e., was not observed. Consequently, file locking was considered to be unnecessary overhead.

Device Drivers

The most interesting part of the project for the Linux hacker is certainly the device driver section. As usual, no device driver could be obtained from manufacturers of the boards to be used. It is a pity that no manufacturer of measurement hardware recognizes the potential of Linux as a measurement platform. Certainly not a real-time system, but with today's fast processors and some precautions, Linux is able to stream data to the disk at high rates and without dropping data.

Writing a device driver seemed to be a daunting task, and it proved to be exactly that, but for reasons other than the expected ones. Not being particularly familiar with the internals of Linux, it first seemed that learning the interface between kernel and driver might be a complicated problem. It proved to be almost too trivial to mention. With an early version of the kernel hacker's guide, code of other drivers all around, and the helpful Net community, communication between kernel and driver was easily established.

The bad part was the hardware, mainly due to a lack of decent documentation. German distributors were approached to almost no avail, even for analog devices. Linux was as yet unheard of, and all that could be obtained was source code for MS-DOS and a user's guide for the RTI-860 containing a full schematic diagram. For the ADCO, the situation was just about the same. Nevertheless drivers were written, and the work is almost perfect today. Only the RTI-860 driver still contains a nasty bug, probably due to a timing problem: clearing the on-board memory and enabling the trigger cannot be done in the right order. Independent of which operation is done first, some samples are sometimes dropped, presumably only if the trigger line goes active very shortly after the trigger is enabled.

Another problem is the kernel itself. This problem was observed in 1.x kernels and seems to persist in 2.0.x kernels. Because the ADCO board has only a one kilometer sample FIFO and must be emptied before it overflows, at a 50KHz sampling rate the driver has to read the data out at a rate of 50Hz. Put another way, the driver has to have a look at the board at least every 20ms. With a time slice of 10ms in a typical Linux kernel, this must happen every other jiffy. For those not familiar with kernel code, it should be noted that there is a variable called a jiffy in the kernel, which is incremented by the timer interrupt. In the Linux kernel, a jiffy is defined rather exactly to be 10ms. In particular with the POSIX scheduler available in recent kernels, this should not present a problem. In contrast to the normal Linux scheduler which constantly changes process priorities to distribute processor time in a fair way, the POSIX scheduler allows a fixed priority to be attached to a process. With the right priorities, at least one process can be guaranteed to get the processor at the next scheduling event after it makes a request. This should be at the next tick of the clock, which is at most 10ms away.

In practice it was found that sometimes no scheduling occured for 40, 50 or even 100ms, which was even more irritating as no other process was active at that time. It looked very much like the mechanism responsible for paging and/

or swapping was responsible for it, but due to limited resources, the problem could not be further investigated.

As a workaround, a mechanism in the kernel was exploited which allows small pieces of code to run between two jiffies. Although no scheduling was performed for up to 100ms, the timer interrupt was not blocked and ticked along fine every 10ms. One of its tasks is to run code which is registered on a certain queue by other parts of the kernel. By registering a function which reads the ADCO's FIFO into a driver-internal buffer, the problem of missing scheduling events could be circumvented. In fact, it is not even necessary to use the POSIX scheduler.

Conclusion, Open Questions and Lessons Learned

Linux proved to be an absolutely stable platform for software development and autonomous data acquisition. The three finger salute (**ctrl-alt-del**), well known on certain widespread desktop program launchers, is never necessary on Linux.

Using A/D conversion boards with on-board memory precludes all real-time constraints. Boards with too little memory are not easily supported. The fact that scheduling is sometimes suppressed for more than 100ms is considered a bug and first resulted in some hectic and active kernel debugging in cooperation with Ingo Molnar (Wien). It turned out that there seemed to be more than one reason for the problems, and they were reported to the kernel developers by Mr. Molnar. However, since we could not wait for the problem to be corrected (a simple patch seemed not to be enough), the solution described above was chosen.

Programming feature-rich A/D conversion boards proved to be more complicated than expected. Even the driver for the well-documented RTI-834 was not easy because of the many dependencies in time and logic between subcomponents of the board. It seems as if a general problem with A/D conversion boards is that designers put too many features on one board introducing dependencies and side effects only they are able to deal with correctly. This might be the reason why it is usually not possible to get good documentation—it simply does not exist, because nobody is able to write it.

A new and very interesting trend in measurement devices was recently initiated by Intelligent Instrumentation (a Burr Brown company). Their EDAS (Ethernet Data Acquistion System) is a 16 channel, 12 bit, 100KHz A/D conversion device which can be hooked to the Ethernet. For UNIX they deliver a library in source code to talk to the device, i.e., program it and read the data. No new device driver must be written. The device can either be connected to a local network or, if continous high speed transfer is necessary, it can be connected to its own “network”--a direct line between the device and a dedicated Ethernet board in the computer. However, while this idea is very nice and is similar to those fashionable WebCams, the EDAS is a bit broken for two reaons: A minor annoyance is that it does not understand RARP (reverse address resolution protocol). To set its IP address, it has to be connected to a computer via a serial line. A more major problem is the device's inability to continuously pump the 100KHz it samples onto the Net. After the first enthusiasm we were very disappointed when the German distributor told us that the EDAS' microcontroller can fill the internal 32 kilometer samples of memory at 100KHz, but that it is too slow to stream the data to the Ethernet at the same speed.

Considering the price of 2500 DM (about $1400 US), it would be cheaper to combine a single-board PC (1000 DM) with an A/D conversion board (1000 DM) and, say, some flash RAM as replacement for a disk into a small case. Install a minimal Linux and a suitable daemon as an interface between IP and the device driver of the A/D board, and you have an iDAB (Internet Data Acquisition Box). Depending on the application, you can even install software to preprocess the data before it is passed to the network.

Harald Kirsch is currently employed at IITB where he managed to convert his group from DOS-based to Linux-based systems. In his free time he works on his degree. If he has free time after work and school, Harald likes to swim, cycle and play volleyball and badminton. He can be reached at kir@iitb.fhg.de.

Archive Index Issue Table of Contents

Advanced search

New Products

Amy Kukuk

Issue #49, May 1998

Visual Prolog 5.0, Samba: Integrating UNIX and Windows, VPN Client and more.

Visual Prolog 5.0

Prolog Development Center has announced the release of their Prolog development environment, Visual Prolog. The new version contains speed improvements, a feature to easily find Runtime errors, project sharing and source control, support for objects and classes, a new linker which can build programs for all platforms without use of a C compiler, Internet support and other miscellaneous improvements. Visual Prolog 5.0 is available for $715 US.

Contact: Prolog Development Center, 568 14th Street, Atlanta, GA 30318,Phone: 800-762-2710, Fax: 404-872-5243, E-mail: sales@pdcatlanta.com,URL: http://www.visual-prolog.com/.

Samba: Integrating UNIX and Windows

Specialized Systems Consultants, Inc. has announced the publication of a new book—*Samba: Integrating UNIX and Windows* by John D. Blair. The book is a combination of technical tutorial, reference guide and how-to manual. It also contains a CD-ROM which contains versions 1.9.17 and 1.9.18alpha of the

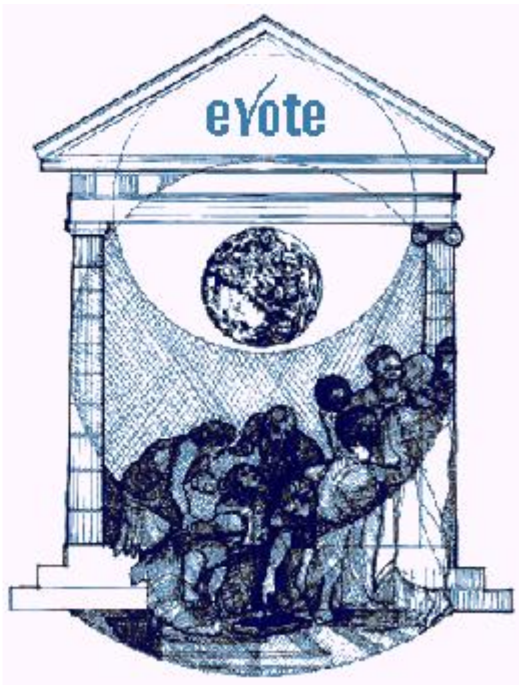
Samba server, a library of tools and scripts and Samba mailing list archives. The price of the book is $29.95 US and can be ordered from Computer Literacy at http://www.clbooks.com/ and is soon to be available in bookstores everywhere.

Contact: Specialized Systems Consultants, Inc., P.O. Box 55549, Seattle, WA 98155-0549, Phone: 206-782-7733, Fax: 206-782-7191,E-mail: info@linuxjournal.com, URL: http://www.ssc.com/.

VPN Client

Aventail Corporation has announced the Aventail VPN server. Corporations can communicate privately, exchange confidential information and share mission-critical applications over the Internet with their suppliers, business partners, customers and remote and mobile employees. Features of the new product include easy installation and administration, TCP/IP transport or device drivers, easy integration into a company's existing network infrastructure, support of multiple authentication and encryption methods including user name/password, CHAP, RADIUS, SSL, Digital Certificates, Token Cards, S/Key, DES, Triple DES, MD4, MD5, SHA-1, RC4 and Diffie-Hellman. The Aventail VPN client for UNIX pricing starts at $7,995 US.

Contact: Aventail Corporation, Phone: 800-762-5785, Fax: 206-215-1120, E-mail: info@aventail.com, URL: http://www.aventail.com/.



eVote 2.2

eVote is a freely available add-on to e-mail list-servers that gives the members of the list the ability to poll each other. After installation of the software, the administrator is not involved. All participants have the power to open polls, vote, change their votes and view each other's votes, if the particular poll was so configured. The underlying specialized data-server, The Clerk, is also freely available for Linux systems only. eVote 2.2 is available in both English and French.

Contact: Marilyn Davis, Phone: 415-493-3631, E-mail: mdavis@deliberate.com, URL: http://www.Deliberate.com/.

Third Beta “Huesten” of KDE

The KDE Core Team has announced the availability of the third public beta “Huesten” of the K Desktop Environment. KDE is a graphical desktop environment for Unix workstations. The KDE desktop aims to combine ease of use, functionality and graphical design. KDE is a new desktop, incorporating a large suite of applications for Unix workstations. KDE includes a window manager, file manager, panel, control center and many other components. Highlights of the new release include support of 18 languages, new applications, kappfinder and improved proxy support. You can download the KDE base packages from ftp://ftp.kde.org/pub/kde/ or one of its many mirrors.

Contact: The KDE Core Team, E-mail: rwilliams@kde.org.

TowerJ 2.0

Tower Technology Corporation has announced the release of TowerJ 2.0 for Linux. TowerJ 2.0 is a high performance compiler and execution environment that takes Java Bytecode as input and creates Linux executables. TowerJ 2.0 is used to improve the performance of server-side applications that have been compiled into 100% Pure Java bytecode and tested using a standard JDK and/or JIT.

Contact: Tower Technology Corporation, 1501 W. Koenig Lane, Austin, TX 78756, Phone: 800-285-5124, Fax: 512-452-1721, E-mail: tower@twr.com,URL: http://www.towerj.com/.

NetVue/JAVA

AccuSoft Corporation has released NetVue/JAVA. NetVue/JAVA is a Java document viewer applet for the Internet/Intranet. NetVue/JAVA provides full platform independence and features full support for displaying annotations, a small footprint with minimized memory usage and scale-to-gray display enhancement for quality and readability. The new release also includes a page thumbnail browser, zooming and scrolling options and TIFF, JPEG and GIF support.

Contact: AccuSoft Corporation, 2 Westborough Business Park, Suite 3013, Westborough, MA 01581, Phone: 508-898-2770, E-mail: sales@accusoft.com, URL: http://www.accusoft.com/.

WipeOut 1.2

Softwarebuero m&b has announced the release of WipeOut 1.2. WipeOut is an integrated software development environment for C++ and Java projects. It contains a project/revision browser, text editor, class browser, make tool, symbol retriever and a debugger front end. The Standard edition is freely available. The price for the Pro edition, which includes revision management and teamwork features, starts at $45 for a single user.

Contact: Softwarebuero m&b, Weststr. 9, 04425 Taucha, Germany, E-mail: info@softwarebuero.de, URL: http://www.softwarebuero.de/.

Archive Index Issue Table of Contents

Advanced search

Advanced search

RAID0 Implementation Under Linux

Jay Munsterman

Issue #49, May 1998

A practical guide to setting up and using a RAID0 device with the multiple device (md) driver.

Most of us who use Linux at home don't have the same requirements as businesses that consider Linux a cost-effective, open alternative to expensive and proprietary Unices. Usually RAID devices aren't a requirement of the home user, although many users running a striped swap partition report a big improvement in speed. The multiple device (md) driver, written by Marc Zyngier, brings RAID to Linux.

md is a driver (included in the standard kernel distribution since 1.3.69) that allows you to group a number of disk partitions together so that they act as a single block device. **md** differs from the other drivers because it doesn't truly access the physical devices that compose it. **md** redirects requests from the upper layer to the devices involved and is interface independent, allowing IDE, SCSI and XT disks to be grouped as a single device.

There are three modes that md can use with its devices: linear, RAID0 and RAID1. In linear mode, the physical devices are appended to each other. When the first device reaches capacity, data is sent to the next device in the group. This mode allows for the creation of a device with a greater capacity but offers no real improvement in performance. RAID0 (or striped) devices spread the data evenly across all the devices in the group. Each write is broken into “chunks”, and the chunks are placed sequentially across the physical devices. RAID0 offers performance improvements, especially with concurrent disk access. RAID1 adds mirroring to RAID0. I feel that RAID0 is the most important of these modes; therefore, it is the focus for the remainder of this article.

Preparation

When planning your RAID0 implementation, there are two considerations to keep in mind: physical device layout and device size. If you use partitions on the same physical device, you will not see any real benefit. The best recommendation I can make is to use several SCSI disks with each partition having the same number of blocks. This seems to offer the best performance. **md** can deal with different size devices as long as there is a significant difference. Using a 1,000,000 block device and a 1,000,001 block device can lead to problems. If you were to create an md device with a 500MB, a 1000MB and a 1500MB partition, it would run fine; md would split the device into “stripe zones” of 500MB. Once 1500MB was written to the device, the first physical device would be full. The second stripe would then be used on the second and third device. After another 1000MB is written, all data would be placed on the last device. Performance decreases in this arrangement as disk usage increases.

Once you have set up the partitions to be used, the kernel will have to be recompiled with md support enabled. Run **make config** (**menuconfig** or **xconfig**) and select “Multiple Device Support” and either “Linear” or “RAID0” mode. Compile as usual. While rebooting with the new kernel, you should receive a message like this:

```
md driver 0.35 MAX_MD_DEV=4, MAX_REAL=8
raid0 personality registered
```

If it went by too fast and you think you may have missed it, use the following command:

```
dmesg | more
```

to receive a replay of the messages logged at boot time. The messages show that md version 0.35 is installed with support for up to four devices, each being made up of up to eight physical devices with RAID0 support. If you think you either will need more than md0 to md3 or will be using more than eight physical devices in an md, the md.h file must be edited prior to compilation; it is usually located in /usr/src/linux/include/linux. Change the value defined for **MAX_REAL** or **MAX_MD_DEV** to fit your requirements.

You now have md support in your kernel, or as a loadable module if you went that way. Next you need to obtain the tools to manage your md devices. Although md is supported in the kernel, it appears that most distributions don't include the tools. They are available from ftp://sweet-smoke.ufr-info-p7.ibp.fr/public/Linux or from the mirror in the U.S. at ftp://linux.nrao.edu/pub/linux/packages/MD-driver. Red Hat software has an RPM distribution available at ftp://ftp.redhat.com/pub/contrib/RPMS. The file md-035-3.i386.rpm contains the needed binaries. Once you have downloaded and unpacked the source, become root and run **make install**. The compilation is straightforward, and I've never had a problem with it. If your Linux source code tree is not located in /usr/src/linux, you will need to edit the Makefile; otherwise, it should compile out of the box.

Creating an MD Device

Now you're ready to actually create a RAID0 device. The compilation created several tools for the task: **mdadd**, **mdrun** and **mdstop**. **mdadd** is used to add block devices to an md device. If you want to use sda1, sdb1 and sdc1, you issue the command:

```
/sbin/mdadd /dev/md0 /dev/sda1 /dev/sdb1 \
        /dev/sdc1
```

This command adds sda1, sdb1 and sdc1 to md0. This same result can also be accomplished by giving these commands:

```
/sbin/mdadd /dev/md0 /dev/sda1
/sbin/mdadd /dev/md0 /dev/sdb1
/sbin/mdadd /dev/md0 /dev/sdc1
```

Remember that the order in which the devices are added is significant. If you change the order, any data previously written will be lost. I recommend adding the devices in what seems like a logical order and then sticking to it.

Now we must start the device. **mdrun** has the following command syntax:

```
/sbin/mdrun -p
```

where *x* indicates the mode: **-l** for linear, **0** for RAID0 and **1** for RAID1. To start the device we just made, the command would be:

```
/sbin/mdrun -p0 /dev/md0
```

When using RAID devices, another option you can use is **-c*n*k** to specify chunk size, where *n* is the chunk size in KB (*n* must be a power of two). For example, **-c6k** indicates a 6KB chunk size. The default value is the value of your **PAGE_SIZE**. The best value for chunk size would be the average request size, so chances are two requests will write to different physical disks. If you plan to use the md for swap space, stick with the default.

Once the device is running, you can create a file system and mount it. For example:

```
/sbin/mkfs.ext2 /dev/md0
mount /dev/md0 /var/spool/news
```

This will create an ext2 file system and then mount it as the news spool. Your RAID0 device is now ready for data. To check its status, type:

```
cat /proc/mdstat
```

and receive the following output:

```
Personalities : [2 raid0]
read_ahead 120 sectors
md0 : active raid0 sda1 sdb1 sdc1 168588 blocks 4k chunks
md1 : inactive
md2 : inactive
md3 : inactive
```

This report tells you which modes are supported, the current read_ahead value, the state of each md device, its mode, physical parts, total size and chunk size.

Managing Your MD Device

At this point we have our RAID device running and mounted; as soon as the machine is rebooted, we will have to rerun mdadd, mdrun and mount. All of this can easily be added to your rc.local file, but there is a better way. **mdcreate** automatically creates an /etc/mdtab file. The mdtab file serves a function similar to the /etc/fstab file, informing the system of the component devices, modes and mount points. The syntax is:

```
mdcreate [-cxk] mode md_dev dev0 dev1 ...
```

To create an mdtab file for our example device we would use:

```
/sbin/mdcreate raid0 /dev/md0 /dev/sda1\
        /dev/sdb1 /dev/sdc1
cat /etc/mdtab
# mdtab entry for /dev/md0:
# /dev/md0  raid0,4k,0,fe8a9ffb  /dev/sda1 /dev/sdb1 /dev/sdc1
```

With this file in place, we can reduce the mdadd command to **mdadd -a** or **mdadd -ar** to automatically add the devices and run them. This also ensures that the devices will always be added in the correct order.

If there is ever a need to stop the device, first unmount it and then use mdstop. **mdstop** will free the physical devices and flush the buffers. For our example device, we would first stop the news server if it was running with the command:

```
/sbin/mdstop /dev/md0
```

Then, we could unmount it using:

```
umount /var/spool/news
```

md0 is now inactive, and the physical partitions can be used elsewhere. Remember, if the device is stopped, none of the data that was written to the md device is accessible.

With md, the implementation and management of RAID devices is made easy. As development continues, we will see RAID1 and the tools necessary for mirror management and recovery. To stay current on the development process, join the Linux-raid mailing list. To subscribe send an email to Majordomo@vger.rutgers.edu with a one line body that says:

```
   subscribe linux-raid <
```

Be sure to look at the documentation that comes with the md package. It's tools like this one that are helping Linux find a place in the business world.

MD at Work

Jay Munsterman has just relocated to Atlanta, GA from Washington DC, where he works with a variety of Unix platforms, Linux being his favorite. In his spare time he likes to spend time with his soon-to-be wife, Denessa, and their dog Melman. Jay can be reached at jmunster@mindspring.com.

KDE and Gnome

Larry Ayers

Issue #49, May 1998

A quick look at two projects designed to make the administration and usage of a Linux system easier for beginners.

Watching the Linux operating system mature is interesting these days. A couple of years ago a lot of attention was devoted to incompatibilities with various hardware components, networking and the development of the kernel. Though these activities continue, it's no longer necessary to follow these development efforts as closely in order to run a dependable Linux system. Distributions have improved immensely, and now more free-software developers are turning their attention towards refinement and integration of the user interface.

Two separate projects have arisen in the past year: KDE (the K Desktop Environment) and GNOME (the Gnu Network Object Model Environment). Both of these projects include among their stated goals the desire to make the administration and usage of a Linux system easier for beginners, in part by providing a uniform look and feel for the most commonly used applications and utilities, as well as interoperability of the system components. It's difficult to make much of a comparison between the two, as KDE is much farther along than GNOME, but I'll make an attempt.

Commonalities

There is one common structural aspect to these two projects. They each rely on a group of shared libraries that provides the interface to basic OS operations, such as file-reading and saving, as well as basic display and appearance functions. An installation will populate a directory with a variety of shared libraries which support a directory of fairly small executables. The Gimp works this way as well; the individual plug-ins tend to be small, but rely on the services provided by both the GTK and the GIMP shared libraries. This approach facilitates contributions by programmers not directly involved with a project, as many of the low-level and window-display functions are already written, allowing a contributed application or extension to “hook” into them.

KDE

The first of the two projects to gain momentum was KDE. About a year ago a group of developers, mainly European, began coding the components of this ambitious project. They chose the Qt toolkit (from TrollTech in Norway) as the GUI framework, a decision which has since led to some controversy. Qt has a few licensing restrictions which, though not onerous for end-users, can cause problems for the creators of CD-ROM-based distributions. Advocates of GNU-style free software tend not to favor Qt, a circumstance which led to the creation of the GNOME project.

Setting aside the thorny licensing issues, the KDE developers have managed to pull together quite a remarkable system in the past year, though numerous bugs still remain evident. The second public beta was released in November of 1997, and I compiled and installed it soon after. (I had briefly tried the initial beta, but found it too unstable to evaluate.)

This second release still has flaky aspects, but enough of it works to give the user an idea of what the developers are planning to accomplish. In effect KDE is a sort of GUI wrapper around an existing Linux system, which attempts to simplify system-administration tasks and offer interactive compatible utilities and applications. **kfm** is at the core of the system, as it is intended to be left running in the background and serves as the help viewer for all of the KDE components. **kfm** is also a file manager (icon-based, with some resemblance to **xfm** and **moxfm**) and serves creditably as a web browser.

kfm is an impressive application, and itself a good reason for trying out KDE. Many of the other applications are replacements for programs which most Linux users probably already have and would only be desirable if a complete KDE system were the goal.

KDE has its own window manager, **kwm**, which had some display faults on my system. Due to these video artifacts I didn't use it much, but it did appear stylish and well-designed. It seems that these display bugs don't show up on most systems; I suspect that it depends upon the video-card and X server being used.

A new Linux user (especially someone accustomed to Windows or Macintosh systems) might appreciate the relative ease of configuration and use which KDE offers. In a sense, KDE extends the scope of the tasks traditional distributions perform. One drawback might be the very comfort of the KDE environment; the various system-administration tasks outside of KDE's abilities might seem too daunting or unapproachable without a KDE interface. This won't be seen as a drawback to prospective users who lack the fascination with internals and configuration which in the past has typified Linux users.

Some KDE users have reported that they find the system both usable and useful, but with my particular setup this wasn't the case. I have to say that my extensively customized Linux installation seems perfectly satisfactory as is, and I probably lack the motivation to spend the time learning to adapt KDE to my needs. If KDE had existed back when I first booted up a Slackware system some years ago, maybe I would have felt differently.

GNOME

Miguel de Icaza, head of the Midnight Commander development group, also seems to be at the helm of the new GNOME development project, which has goals similar to those of KDE, with one difference: the project is composed completely of GNU-style free software. This project is based upon the GTK toolkit, the free successor to Motif in the GIMP development efforts. The project arose as a direct response to the KDE efforts, and the GNOME developers have borrowed some code from KDE for a few of the applets.

As of mid-January (version 0.12) GNOME as a whole isn't really suitable for actual use, but several of the applets function well and the future looks bright for the project. Miguel de Icaza is in the process of porting the Midnight Commander file-manager to GTK, which will allow it to fit in with the remainder of the GNOME applications.

The Panel applet, written primarily by Federico Meña Quintero, is an icon-bar and program-launcher which is located at the bottom edge of the screen. It features cascading menus which could be a substitute for the usual window-manager root menus. Most of the GNOME applets have been included in the default menu of Panel, allowing this applet to serve as an entry-point to the GNOME installation. It takes a little fiddling around to get the hang of using Panel, so don't give up if at first glance it seems like nothing is working.

The provided applets include a desktop manager (which in part serves as an interface to the Xlockmore screensaver), CroMagnon (an interface to the **crontab** utility), an audio mixer, an interface to the elaborate LinuxConf configuration manager, several nicely-done games (some of which were adapted from KDE), a calculator and several others.

One major difference between GNOME and KDE is that KDE includes a window manager, whereas GNOME doesn't. GNOME is designed to cooperate with the user's current window manager. This may make GNOME more appealing to seasoned users who have extensively customized their window-manager resource files.

Conclusion

As I write this, only the source code is available for GNOME 0.12, and it's tricky to compile. Several GNU utilities, such as gettext, guile and SLIB, must be correctly installed in order for a compilation to complete successfully. An intel-Linux binary archive of the 0.9 release is available from ftp://ftp.nuclecu.unam.mx/GNOME, but I would recommend waiting a while for either an updated binary release or an easier-to-build source release. Another drawback is the lack of any man pages or help files. The developers are hard at work these days (judging by their mailing-list postings), and I think, given time, that something both interesting and usable will appear.

Though KDE is closer to being “finished” (if such a state even exists in the realm of software), it still has a way to go. Development is proceeding rapidly, and I imagine that sometime this year a more polished release will become available.

The fate of a free-software project is interesting because of the inherent unpredictability. Anyone can start one, but whether it comes to fruition or withers on the vine is up to the inscrutable software gods. The timing may be just right (i.e., the software addresses many users' and developers' needs), but convincing enough programmers with time and inclination to become involved just can't be forced or foretold. These two projects seem to have attained that essential momentum, and hopefully we shall see them evolve further.

Larry Ayers (layers@marktwain.net.) lives on a small farm in northern Missouri, where he raises sheep, shiitake and shell-scripts.

Archive Index Issue Table of Contents

Advanced search

Best of Technical Support

Various

Issue #49, May 1998

Our experts answer your technical questions.

Viruses

I would like to know what can be said about viruses in a Linux system which is installed on the same hard disk as Win95. For example, what can happen if a virus infects the MBR sector (where resides LILO) or if I mount an infected MS-DOS formatted diskette?

—Troha Donato

Leaving out the usual statements about Unix systems being immune to standard virus attacks, this is an important point most people should consider, since many people who run Linux on personal computers (as opposed to servers) also run Win95 or some other operating system. You should be safe from a mounted floppy, but be warned that you can get some very odd effects reading such a disk, such as strange directory entries.

There are several scenarios, from boot sector infection to random pot shots some viruses are known to take. Unlike the DOS file systems, which concentrate their layout information into one or two dense tables, Linux spreads these across the disk. Random potshots are much more likely to wipe out vital structures on a Linux disk than they are on a DOS disk (assuming the virus ran from a booted DOS system).

Safety first, as always. When in Unix, don't use the root user account unless you need to. Create a normal user account in which to do your work. When in DOS, scan—scan—scan.

—Chad Robinson, BRT Technologies Senior System Analyst chadr@brt.com

Chances are that if your system becomes infected with a boot sector virus, LILO will no longer work. The best defense against this situation is to keep an emergency boot floppy handy. I generally create them using the command **dd if=/vmlinuz of=/dev/fd0**. You will want to write-protect it of course. After booting from floppy, simply re-run LILO. Assuming you have LILO configured to use the system MBR, it will overwrite the virus.

The Linux operating system itself is not very vulnerable to MS-DOS-style viruses. All of the common ones depend on being in a DOS/Windows environment. They do not know how to cope with Linux and do not function.

—Keith Stevenson k.stevenson@louisville.edu

Memory Allocation

I have been working with the Linux system for almost two years. My problem is memory allocation. The kernel (2.xx) does not reclaim memory after things such as X sessions are perfomed. I am constantly rebooting the machine (**shutdown -r now**) in order to gain sufficient memory for multiple operations. Is there an executable that can be run which will free all possible memory that current kernel processes are not using?

If the answer is no, then how can one use Linux as an httpd server that takes a lot of hits per day? The system would almost always be short of memory to be able to quickly service, multiple httpd server and other processes. In short, I am somewhat disappointed in the way Linux handles memory reclamation. Is it that the X Server and applications are simply “poorly written” and do not free memory upon exit?

—George R. Boyko

There are no memory leakages in Linux 2.0; there may be some in the 2.1 kernel series, but those versions are only beta-releases aimed specifically at developers. It's true, on the other hand, that the amount of free memory reported by a running Linux system is always tiny. This is a feature rather than a bug; free memory is just wasted, and Linux tries to avoid any waste by keeping disk buffers and page caches in an otherwise waste-free memory.

It's the kernel which releases any process resources upon exit. You don't want your students to lock memory by not calling **free**, do you? As a matter of fact, many one-shot programs are “poorly written” and rely on the system to close files and release memory.

—Alessandro Rubini rubini@linux.it

Memory management is one of the things I really like about Linux. I find it to be much more efficient than a certain popular commercial OS.

I have several Linux systems, all with 64MB of installed RAM. I use **xosview** to monitor things like CPU activity and memory utilization. These machines function as ftp servers, web servers and multi-user workstations. According to xosview, the memory utilization is consistently above 90% even when the machine is lightly loaded. This isn't a problem. It simply means that there is a lot of stuff cached in memory. The real indicator of whether or not you have enough RAM in your system is swap space utilization. This can be monitored with xosview or with the command **vmstat**. If you are swapping to disk often, you probably need to add more RAM to the system. If not, then things are probably okay. My 64MB systems almost never swap out to disk, and they have excellent response time despite the fact that 90% or more of their RAM is marked as being “in use”.

—Keith Stevenson k.stevenson@louisville.edu

Geometry Mismatch Error

I am having a problem with LILO. It hangs after the letters “LI”. I read the MINI-HOWTO, and it says that the first boot loader was able to load the second boot loader but has failed to execute it. Then it goes on to say that the cause is a “geometry mismatch”. Any suggestions?

—Jim Mendoza Red Hat 4.2

LILO loads its second-stage loader and then the kernel by accessing disk blocks based on their disk location (CHS: Cylinder, Head, Sector). A “geometry mismatch” is what happens when LILO's map uses CHS values that are not those used by the BIOS; this happens with modern BIOSes that play dirty games with disk geometry to overcome a limitation built in Microsoft programs. Add a “disk =” section to your /etc/lilo.conf to specify disk geometry as Linux sees it.

—Alessandro Rubini rubini@linux.it

Undetected Modem

Linux does not detect my modem at com4 (/dev/cua3, address as **0x02e8**) which works fine in Win95. Each time I reboot the system, it automatically detects only serial port number 1 (/dev/cua0, at **0x03f8**) and port number 2 (/dev/cua1, at **0x02f8**). My modem is internal, non-plug-and-play, 33.6Kbps and manufactured by PC tel.

—Jianzhong Ding Red Hat 4.2

Use **setserial** to tell the serial driver about the location of your ports. “Plug-and-play” is an ugly specification, and most of the time it creates problems. To look for your PnP devices and configure them, run the **isapnp** package.

—Alessandro Rubini rubini@linux.it

Sendmail Pause

During startup, there is a long pause while **sendmail** starts. I can only assume that a request is timing out while trying to contact something on the network (the network, of course, isn't up yet).

Is there a way to shorten the time-out period for sendmail or otherwise correct the situation?

—David Moulton Red Hat 4.0

This may be a problem with your machine name in the /etc/hosts file. Recent versions of sendmail need your name to be a FQDN (including a domain name):

```
   192.168.1.1 foo foo.bar.com
```

If your name is not fully qualified, sendmail will sleep for about one minute.

—Pierre Ficheux, Lectra Systèmes pierre@rd.lectra.fr

The pause is most likely a name server lookup that is timing out. Have a look in your maillog (probably /var/log/maillog) and search for lines that look like these two:

```
   Dec 21 18:33:46 keiko sendmail[4547]:
    gethostbyaddr() failed for 192.168.0.1
   Dec 21 18:33:47 keiko sendmail[4553]:
    starting daemon (8.8.5): SMTP+queueing@00:05:00
```

What's happening is sendmail is trying to resolve the IP address of the machine it's running on. Name server calls take a relatively long time to timeout, thus the delay you are experiencing. The quick solution is to add an entry for this IP address into /etc/hosts or into your name server configuration. **sendmail** starts very quickly after you have done this.

—Keith Stevenson k.stevenson@louisville.edu

Archive Index Issue Table of Contents

Advanced search