Creating and Using Patches to OSS

Most software developers who work in the open source and free software communities will at some point need to exchange modifications to source code. Most people first encounter this when they need to update a piece of software they have compiled from source. The update will invariably arrive in a *patchfile*, and the developer is expected to know what to do with it. Alternatively a developer might find a problem with some software, then examine the source code and work out a fix. The code maintainer will request a patch for the bug fix and under these circumstances the developer would be expected to know how to generate such a patch.

Dealing with patches, either as creator or consumer, is considered a routine part of working in the open source and free software fields, but is often completely alien to those entering the community from the proprietary software world. This article introduces the tools required to work with patches – *patch* and *diff*.

patch is a tool that takes a *patchfile* – the details of the modifications – and applies them to the original version of some code in order to create a new, updated version. Before we look at *patch*, however, in order to understand the complete picture we must start by looking at another tool, called *diff*.

diff is short for difference, and as the name implies its purpose is to report the differences between two files. *diff* actually works on any text file, not just program source files, and so is a very useful tool in its own right. In the context of distributing source code updates, *diff* is asked to report the differences between the old, original version of a source file and the new, modified version, then the output is captured and saved to a file known as a *patchfile*. A patchfile can be read by the *patch* tool which understands how to take a set of differences and apply them to the original source file, thus ending up with a copy of the new source file. We'll look at the use of *patch* in a moment, but first let's look at a few of the details involved in using *diff*.

Using *diff*

diff is a powerful tool in its own right, but for the purposes of generating patchfiles only a small subset of its features are required. In the majority of cases, and unless a project maintainer specifically asks otherwise, diff should be used to create a patchfile in what is known as *unified diff* format.

Unified diff format contains more information than the *normal diff* format. In particular, each difference is shown with a few lines of the surrounding unchanged text above and below it. These *context* lines allow a reader (machine or human) to more easily work out what has been changed. A unified diff patchfile is created by supplying the –u flag to the *diff* command like this:

```
diff –u file_old.c file_new.c > patchfile
```

The two files are given on the command line, with the original file first and the new, edited version second. It isn't really necessary to understand the format of the diff output (i.e. the contents of a patchfile), either to create them or use them, but a basic understanding of the format can be useful to check things look right. Applying the above example to a file containing one digit per line (on the left of the figure below) and an edited version of it (on the right):

```
        1                                       1
        2                                       2
        3                                       3
        4                                       4
        5                                     five
        6                                     6
        7                                     7
        8                                     8
        9                                     9
```

results in this patchfile:

```
--- numbers.txt.orig    2004-05-28 18:08:55.000000000 +0800
+++ numbers.txt 2004-05-27 15:16:18.000000000 +0800
@@ -2,7 +2,7 @@
 2
 3
 4
-5
+five
 6
 7
 8
```

The line containing --- shows the name of the first file – the original text. The line containing +++ shows the name of the second file – the modified text. The modified file should take the real filename, as it will appear on the patch user's system. The line that starts @@ shows the line numbers where the modification has happened, and below that is a block – known as a *hunk* - which shows the actual change. A line starting with a '–' indicates that the line appears in the original file but doesn't appear in the modified file. A line starting with a '+' indicates that the line wasn't in the original file, but now appears in the modified file. When one or more '–' lines appear followed by one or more '+' lines, like in this example, it means some text has been removed and replaced. In this case I took out the line containing '5' and replaced it with a line containing 'five'. The other lines, containing the numbers 2, 3 and 4, and 6, 7 and 8, are the *context* lines. These are the lines on either side of the change which weren't modified. If you're familiar with the source file these lines help make the patchfile more readable.
This is obviously a very simple example. Real patchfiles almost always detail large sections of code which have been removed, added and replaced, and there might be dozens of hunks per patchfile. A real patchfile is often an alarming sight to the inexperienced, but don't be intimidated by them. As indicated above, it is rare that a human needs to really understand a patchfile. As long as a quick skim read shows that the hunks reflect changes to the right files, and that a 3 line change hasn't somehow resulted in a 300K patchfile, all that matters is that the *patch* tool understands the details.

Using patch

patch is the other half of the *diff/patch* pairing, and is the tool used by the patchfile 'consumer'. *patch* will take a copy of the original file, examine the difference information in the patchfile, and will then apply those differences as changes to the original file resulting in a copy of the modified file. As we've seen, the *unified diff* file format describes the changes in compete detail – what has been removed, what has been added, and the line numbers where the modifications take place.
patch is smart enough to notice when the line numbers it finds in the patchfile don't match up with the lines that have been reportedly removed or added. This might happen if the source file the patch was created from isn't identical to the source file the patch is being applied to, perhaps because work has already been performed on

the local code. In these cases *patch* can use the context lines around each hunk to try to pin down exactly where the change needs to be applied. Obviously this system isn't foolproof, but if the changes that have been made to the local source file aren't too severe, *patch* will often succeed in patching an already modified file. This is very useful if you receive two patches to the same file at the same time – they can be applied consecutively and there's a fair chance they will both succeed.
patch takes the patchfile on its standard input, so given the patchfile created in the example above, and a copy of the original numbers.txt file in the current directory, the edited version of the file can be obtained with this command:

```
> patch numbers.txt < patchfile
patching file numbers.txt
```

The filename is optional; if it is not specified *patch* will look in the patchfile to find the name of the local file to patch based on the names of the files on the machine where the patchfile was created. The heuristics used to find the name of the target file are pretty good for most uses of patch, so it is normal to use the simplest patch command:

```
> patch < patchfile
```

By default the patching happens "in place", so to speak. Check the *patch* man page for details on how to send the patched output to a different file, or to have backup files produced automatically.

Patching Trees

Patching one file is easy, and is the accepted way to fix a simple bug or typo. A more commonly experienced process is that of updating an entire software tree. When a new version of a project is made available it is normal for the software maintainer to provide a complete tarball of the whole source tree, together with a single patchfile that can be applied to the old source tree to bring it up to the latest version. Anyone who wants to start working with the project would download and unpack the tarball; anyone who already has the previous version of the software only need to download and apply the patchfile. patchfiles can be applied consecutively, so if your source tree is three versions behind the bleeding edge, you can download and apply the last three patchfiles one after the other to come up to date. This is almost always faster than downloading a whole new tarball.
As a patchfile creator, you should put the original and modified trees in the same directory, then tell *diff* to operate recursively to produce a patchfile that encompasses all the differences between the old files and the new ones. So, if I have just completed the work to bring my project up to version 1.1, I need both the 1.0 and the 1.1 trees in one subdirectory:

```
> pwd
/home/derek/project
> ls -l
drwxr-xr-x    2 derek    users           48 2004-05-20 17:55 1.0
drwxr-xr-x    2 derek    users           48 2004-05-28 11:49 1.1
```

To produce the patchfile I need to run *diff* in recursive mode, with the root subdirectories as the parameters:

```
> diff -urN 1.0 1.1 > patchfile
```

The –r flag invokes the recursive operation, and the –N flag makes *diff* treat files which are in one tree but not the other as new files (which allows *patch* to handle file creation and deletion neatly). Although this is the most standard way of creating a patchfile for a source tree, there are many options to *diff* which are useful in special cases. These include options to ignore certain files, handle whitespace efficiently (so simple differences in the use of tabs and spaces don't create huge patchfiles), and other more fundamental things, such as changing the heuristics of the difference detection algorithm to favour, for example, large files. See the *diff* documentation for details.
A patchfile for a source tree is applied by changing to the directory containing the root of the original tree and using the command:

```
> patch –p0 < patchfile
```

The *patch* tool will automatically recognise that the patchfile contains a whole stream of differences which it needs to apply over the entire tree.
The *patch* –p switch is one that is often required when patching entire source trees. By default *patch* looks at the complete filename given in the patchfile, strips off all the subdirectory part, then looks in the current directory for that file. For individual files this approach works, but for patches containing alterations to lots of files in a source tree it doesn't. The –p switch controls how much of the subdirectory part of the filename is stripped off, with –p0 meaning none of it. With –p0, all filenames are taken from the patchfile in their entirety, which means as long as the directory names in the local tree match those on the machine where the patch was created, the patch will apply correctly. Unless you've renamed the directories in your source tree, -p0 is all you need. You might, however, have your local source tree under a subdirectory called, for example, `project-latest`, but the patch might have been created on a system where the source tree was under a subdirectory named `project-1.1`. The patchfile will therefore contain references to filenames which won't be found on your system and patching with –p0 won't work. In this case you provide a positive number to the –p switch, and it makes *patch* ignore that number of levels of subdirectory within the filenames in the patchfile. For example, if a filename is detailed in the patchfile as `project-1.1/src/gui/main.cpp`, and you use the switch `–p1`, *patch* will ignore the first part of the filename so it becomes `src/gui/main.cpp`. Therefore you can change into your `project-latest` directory and use the command:

```
> patch –p1 < patchfile
```

and the patch will work correctly.
patch tries hard to make your patchfile work. If it can't find a particular file, it will ask where that file is. If it finds a file that appears to have already been patched, it will ask if you want reverse the patch – i.e. to back out the change and put the file back as it was. If there is leading or trailing garbage or indenting in your patchfile, *patch* will ignore it, which means you can save the contents of an email that contains a patchfile and apply it directly. If all these processes fail and part of a patchfile doesn't work (maybe because you have made large changes to one of your source files), *patch* will create a *rejection file*. This is a file with a .rej suffix which contains the hunk that couldn't be used. This is useful with a large patchfile where most of the hunks work correctly, because it leaves you with the details of the piece that didn't work. A developer who is familiar with the structure of patchfiles can use the information in the rejection file to finish the patching job by hand.

Conclusion

patch and *diff* are staple tools of the open source and free software communities. They are used by virtually all developers who generate or apply software updates, either directly from the command line, or indirectly though wrapping tools like CVS. They are powerful, yet simple to understand and use, so anyone who wants to contribute code only needs to spend a short time learning to use them. They are an excellent solution to the most basic problem of distributed software development.

Bio

Derek Fountain is a freelance writer and software developer specialising in Linux and open source scripting languages. He lives in Perth, Western Australia.