Applying Patches To The Linux Kernel

Original by: Jesper Juhl, August 2005
Last update: 2006-01-05

A frequently asked question on the Linux Kernel Mailing List is how to apply a patch to the kernel or, more specifically, what base kernel a patch for one of the many trees/branches should be applied to. Hopefully this document will explain this to you.

In addition to explaining how to apply and revert patches, a brief description of the different kernel trees (and examples of how to apply their specific patches) is also provided.

What is a patch?

A patch is a small text document containing a delta of changes between two different versions of a source tree. Patches are created with the `diff' program.
To correctly apply a patch you need to know what base it was generated from and what new version the patch will change the source tree into. These should both be present in the patch file metadata or be possible to deduce from the filename.

How do I apply or revert a patch?

You apply a patch with the `patch' program. The patch program reads a diff (or patch) file and makes the changes to the source tree described in it.

Patches for the Linux kernel are generated relative to the parent directory holding the kernel source dir.

This means that paths to files inside the patch file contain the name of the kernel source directories it was generated against (or some other directory names like "a/" and "b/").
Since this is unlikely to match the name of the kernel source dir on your local machine (but is often useful info to see what version an otherwise unlabeled patch was generated against) you should change into your kernel source directory and then strip the first element of the path from filenames in the patch file when applying it (the -p1 argument to `patch' does this).

To revert a previously applied patch, use the -R argument to patch.
So, if you applied a patch like this:

```
patch -p1 < ../patch-x.y.z
```

You can revert (undo) it like this:

```
patch -R -p1 < ../patch-x.y.z
```

How do I feed a patch/diff file to `patch'?

This (as usual with Linux and other UNIX like operating systems) can be done in several different ways.
In all the examples below I feed the file (in uncompressed form) to patch via stdin using the following syntax:

```
patch -p1 < path/to/patch-x.y.z
```

If you just want to be able to follow the examples below and don't want to know of more than one way to use patch, then you can stop reading this section here.

Patch can also get the name of the file to use via the -i argument, like this:

```
patch -p1 -i path/to/patch-x.y.z
```

If your patch file is compressed with gzip or bzip2 and you don't want to uncompress it before applying it, then you can feed it to patch like this instead:

```
zcat path/to/patch-x.y.z.gz | patch -p1
bzcat path/to/patch-x.y.z.bz2 | patch -p1
```

If you wish to uncompress the patch file by hand first before applying it (what I assume you've done in the examples below), then you simply run gunzip or bunzip2 on the file -- like this:

```
gunzip patch-x.y.z.gz
bunzip2 patch-x.y.z.bz2
```

Which will leave you with a plain text patch-x.y.z file that you can feed to patch via stdin or the -i argument, as you prefer.

A few other nice arguments for patch are -s which causes patch to be silent except for errors which is nice to prevent errors from scrolling out of the screen too fast, and --dry-run which causes patch to just print a listing of what would happen, but doesn't actually make any changes. Finally --verbose tells patch to print more information about the work being done.

Common errors when patching

When patch applies a patch file it attempts to verify the sanity of the file in different ways.
Checking that the file looks like a valid patch file & checking the code around the bits being modified matches the context provided in the patch are just two of the basic sanity checks patch does.

If patch encounters something that doesn't look quite right it has two options. It can either refuse to apply the changes and abort or it can try to find a way to make the patch apply with a few minor changes.

One example of something that's not 'quite right' that patch will attempt to fix up is if all the context matches, the lines being changed match, but the line numbers are different. This can happen, for example, if the patch makes a change in the middle of the file but for some reasons a few lines have been added or removed near the beginning of the file. In that case everything looks good it has just moved up or down a bit, and patch will usually adjust the line numbers and apply the patch.

Whenever patch applies a patch that it had to modify a bit to make it fit it'll tell you about it by saying the patch applied with 'fuzz'.
You should be wary of such changes since even though patch probably got it right it doesn't /always/ get it right, and the result will sometimes be wrong.

When patch encounters a change that it can't fix up with fuzz it rejects it outright and leaves a file with a .rej extension (a reject file). You can read this file to see exactly what change couldn't be applied, so you can go fix it up by hand if you wish.

If you don't have any third-party patches applied to your kernel source, but only patches from kernel.org and you apply the patches in the correct order, and have made no modifications yourself to the source files, then you should never see a fuzz or reject message from patch. If you do see such messages anyway, then there's a high risk that either your local source tree or the patch file is corrupted in some way. In that case you should probably try re-downloading the patch and if things are still not OK then you'd be advised to start with a fresh tree downloaded in full from kernel.org.

Let's look a bit more at some of the messages patch can produce.

If patch stops and presents a "File to patch:" prompt, then patch could not find a file to be patched. Most likely you forgot to specify -p1 or you are in the wrong directory. Less often, you'll find patches that need to be applied with -p0 instead of -p1 (reading the patch file should reveal if this is the case -- if so, then this is an error by the person who created the patch but is not fatal).

If you get "Hunk #2 succeeded at 1887 with fuzz 2 (offset 7 lines)." or a message similar to that, then it means that patch had to adjust the location of the change (in this example it needed to move 7 lines from where it expected to make the change to make it fit).
The resulting file may or may not be OK, depending on the reason the file was different than expected.
This often happens if you try to apply a patch that was generated against a different kernel version than the one you are trying to patch.

If you get a message like "Hunk #3 FAILED at 2387.", then it means that the patch could not be applied correctly and the patch program was unable to fuzz its way through. This will generate a .rej file with the change that caused the patch to fail and also a .orig file showing you the original content that couldn't be changed.

If you get "Reversed (or previously applied) patch detected! Assume -R? [n]" then patch detected that the change contained in the patch seems to have already been made.
If you actually did apply this patch previously and you just re-applied it in error, then just say [n]o and abort this patch. If you applied this patch previously and actually intended to revert it, but forgot to specify -R, then you can say [y]es here to make patch revert it for you.
This can also happen if the creator of the patch reversed the source and destination directories when creating the patch, and in that case reverting the patch will in fact apply it.

A message similar to "patch: **** unexpected end of file in patch" or "patch

unexpectedly ends in middle of line" means that patch could make no sense of the file you fed to it. Either your download is broken, you tried to feed patch a compressed patch file without uncompressing it first, or the patch file that you are using has been mangled by a mail client or mail transfer agent along the way somewhere, e.g., by splitting a long line into two lines. Often these warnings can easily be fixed by joining (concatenating) the two lines that had been split.

As I already mentioned above, these errors should never happen if you apply a patch from kernel.org to the correct version of an unmodified source tree. So if you get these errors with kernel.org patches then you should probably assume that either your patch file or your tree is broken and I'd advise you to start over with a fresh download of a full kernel tree and the patch you wish to apply.

Are there any alternatives to `patch'?

Yes there are alternatives.

You can use the `interdiff' program (http://cyberelk.net/tim/patchutils/) to generate a patch representing the differences between two patches and then apply the result.
This will let you move from something like 2.6.12.2 to 2.6.12.3 in a single step. The -z flag to interdiff will even let you feed it patches in gzip or bzip2 compressed form directly without the use of zcat or bzcat or manual decompression.

Here's how you'd go from 2.6.12.2 to 2.6.12.3 in a single step:
```
interdiff -z ../patch-2.6.12.2.bz2 ../patch-2.6.12.3.gz | patch -p1
```

Although interdiff may save you a step or two you are generally advised to do the additional steps since interdiff can get things wrong in some cases.

Another alternative is `ketchup', which is a python script for automatic downloading and applying of patches (http://www.selenic.com/ketchup/).

Other nice tools are diffstat, which shows a summary of changes made by a patch; lsdiff, which displays a short listing of affected files in a patch file, along with (optionally) the line numbers of the start of each patch; and grepdiff, which displays a list of the files modified by a patch where the patch contains a given regular expression.

Where can I download the patches?

The patches are available at http://kernel.org/
Most recent patches are linked from the front page, but they also have specific homes.

The 2.6.x.y (-stable) and 2.6.x patches live at
ftp://ftp.kernel.org/pub/linux/kernel/v2.6/

The -rc patches live at
ftp://ftp.kernel.org/pub/linux/kernel/v2.6/testing/

The -git patches live at
 ftp://ftp.kernel.org/pub/linux/kernel/v2.6/snapshots/

The -mm kernels live at
 ftp://ftp.kernel.org/pub/linux/kernel/people/akpm/patches/2.6/

In place of ftp.kernel.org you can use ftp.cc.kernel.org, where cc is a country code. This way you'll be downloading from a mirror site that's most likely geographically closer to you, resulting in faster downloads for you, less bandwidth used globally and less load on the main kernel.org servers -- these are good things, so do use mirrors when possible.

The 2.6.x kernels

 These are the base stable releases released by Linus. The highest numbered release is the most recent.

If regressions or other serious flaws are found, then a -stable fix patch will be released (see below) on top of this base. Once a new 2.6.x base kernel is released, a patch is made available that is a delta between the previous 2.6.x kernel and the new one.

To apply a patch moving from 2.6.11 to 2.6.12, you'd do the following (note that such patches do *NOT* apply on top of 2.6.x.y kernels but on top of the base 2.6.x kernel -- if you need to move from 2.6.x.y to 2.6.x+1 you need to first revert the 2.6.x.y patch).

Here are some examples:

```
# moving from 2.6.11 to 2.6.12
$ cd ~/linux-2.6.11                   # change to kernel source dir
$ patch -p1 < ../patch-2.6.12         # apply the 2.6.12 patch
$ cd ..
$ mv linux-2.6.11 linux-2.6.12        # rename source dir

# moving from 2.6.11.1 to 2.6.12
$ cd ~/linux-2.6.11.1                 # change to kernel source dir
$ patch -p1 -R < ../patch-2.6.11.1    # revert the 2.6.11.1 patch
                                      # source dir is now 2.6.11
$ patch -p1 < ../patch-2.6.12         # apply new 2.6.12 patch
$ cd ..
$ mv linux-2.6.11.1 linux-2.6.12              # rename source dir
```

The 2.6.x.y kernels

 Kernels with 4-digit versions are -stable kernels. They contain small(ish) critical fixes for security problems or significant regressions discovered in a given 2.6.x kernel.

This is the recommended branch for users who want the most recent stable kernel and are not interested in helping test development/experimental versions.

If no 2.6.x.y kernel is available, then the highest numbered 2.6.x kernel is

the current stable kernel.

note: the -stable team usually do make incremental patches available as well as patches against the latest mainline release, but I only cover the non-incremental ones below. The incremental ones can be found at ftp://ftp.kernel.org/pub/linux/kernel/v2.6/incr/

These patches are not incremental, meaning that for example the 2.6.12.3 patch does not apply on top of the 2.6.12.2 kernel source, but rather on top of the base 2.6.12 kernel source .
So, in order to apply the 2.6.12.3 patch to your existing 2.6.12.2 kernel source you have to first back out the 2.6.12.2 patch (so you are left with a base 2.6.12 kernel source) and then apply the new 2.6.12.3 patch.

Here's a small example:

```
$ cd ~/linux-2.6.12.2                  # change into the kernel source dir
$ patch -p1 -R < ../patch-2.6.12.2     # revert the 2.6.12.2 patch
$ patch -p1 < ../patch-2.6.12.3        # apply the new 2.6.12.3 patch
$ cd ..
$ mv linux-2.6.12.2 linux-2.6.12.3     # rename the kernel source dir
```

The -rc kernels

These are release-candidate kernels. These are development kernels released by Linus whenever he deems the current git (the kernel's source management tool) tree to be in a reasonably sane state adequate for testing.

These kernels are not stable and you should expect occasional breakage if you intend to run them. This is however the most stable of the main development branches and is also what will eventually turn into the next stable kernel, so it is important that it be tested by as many people as possible.

This is a good branch to run for people who want to help out testing development kernels but do not want to run some of the really experimental stuff (such people should see the sections about -git and -mm kernels below).

The -rc patches are not incremental, they apply to a base 2.6.x kernel, just like the 2.6.x.y patches described above. The kernel version before the -rcN suffix denotes the version of the kernel that this -rc kernel will eventually turn into.
So, 2.6.13-rc5 means that this is the fifth release candidate for the 2.6.13 kernel and the patch should be applied on top of the 2.6.12 kernel source.

Here are 3 examples of how to apply these patches:

```
# first an example of moving from 2.6.12 to 2.6.13-rc3
$ cd ~/linux-2.6.12                    # change into the 2.6.12 source dir
$ patch -p1 < ../patch-2.6.13-rc3      # apply the 2.6.13-rc3 patch
$ cd ..
$ mv linux-2.6.12 linux-2.6.13-rc3     # rename the source dir

# now let's move from 2.6.13-rc3 to 2.6.13-rc5
$ cd ~/linux-2.6.13-rc3                # change into the 2.6.13-rc3 dir
```

```
$ patch -p1 -R < ../patch-2.6.13-rc3    # revert the 2.6.13-rc3 patch
$ patch -p1 < ../patch-2.6.13-rc5       # apply the new 2.6.13-rc5 patch
$ cd ..
$ mv linux-2.6.13-rc3 linux-2.6.13-rc5  # rename the source dir

# finally let's try and move from 2.6.12.3 to 2.6.13-rc5
$ cd ~/linux-2.6.12.3                   # change to the kernel source dir
$ patch -p1 -R < ../patch-2.6.12.3      # revert the 2.6.12.3 patch
$ patch -p1 < ../patch-2.6.13-rc5       # apply new 2.6.13-rc5 patch
$ cd ..
$ mv linux-2.6.12.3 linux-2.6.13-rc5    # rename the kernel source dir
```

The -git kernels

These are daily snapshots of Linus' kernel tree (managed in a git repository, hence the name).

These patches are usually released daily and represent the current state of Linus's tree. They are more experimental than -rc kernels since they are generated automatically without even a cursory glance to see if they are sane.

-git patches are not incremental and apply either to a base 2.6.x kernel or a base 2.6.x-rc kernel -- you can see which from their name.
A patch named 2.6.12-git1 applies to the 2.6.12 kernel source and a patch named 2.6.13-rc3-git2 applies to the source of the 2.6.13-rc3 kernel.

Here are some examples of how to apply these patches:

```
# moving from 2.6.12 to 2.6.12-git1
$ cd ~/linux-2.6.12                     # change to the kernel source dir
$ patch -p1 < ../patch-2.6.12-git1      # apply the 2.6.12-git1 patch
$ cd ..
$ mv linux-2.6.12 linux-2.6.12-git1     # rename the kernel source dir

# moving from 2.6.12-git1 to 2.6.13-rc2-git3
$ cd ~/linux-2.6.12-git1                # change to the kernel source dir
$ patch -p1 -R < ../patch-2.6.12-git1   # revert the 2.6.12-git1 patch
                                        # we now have a 2.6.12 kernel
$ patch -p1 < ../patch-2.6.13-rc2       # apply the 2.6.13-rc2 patch
                                        # the kernel is now 2.6.13-rc2
$ patch -p1 < ../patch-2.6.13-rc2-git3  # apply the 2.6.13-rc2-git3 patch
                                        # the kernel is now 2.6.13-rc2-git3
$ cd ..
$ mv linux-2.6.12-git1 linux-2.6.13-rc2-git3    # rename source dir
```

The -mm kernels

These are experimental kernels released by Andrew Morton.

The -mm tree serves as a sort of proving ground for new features and other experimental patches.
Once a patch has proved its worth in -mm for a while Andrew pushes it on to Linus for inclusion in mainline.

Although it's encouraged that patches flow to Linus via the -mm tree, this is not always enforced.
Subsystem maintainers (or individuals) sometimes push their patches directly to Linus, even though (or after) they have been merged and tested in -mm (or sometimes even without prior testing in -mm).

You should generally strive to get your patches into mainline via -mm to ensure maximum testing.

This branch is in constant flux and contains many experimental features, a lot of debugging patches not appropriate for mainline etc., and is the most experimental of the branches described in this document.

These kernels are not appropriate for use on systems that are supposed to be stable and they are more risky to run than any of the other branches (make sure you have up-to-date backups -- that goes for any experimental kernel but even more so for -mm kernels).

These kernels in addition to all the other experimental patches they contain usually also contain any changes in the mainline -git kernels available at the time of release.

Testing of -mm kernels is greatly appreciated since the whole point of the tree is to weed out regressions, crashes, data corruption bugs, build breakage (and any other bug in general) before changes are merged into the more stable mainline Linus tree.
But testers of -mm should be aware that breakage in this tree is more common than in any other tree.

The -mm kernels are not released on a fixed schedule, but usually a few -mm kernels are released in between each -rc kernel (1 to 3 is common).
The -mm kernels apply to either a base 2.6.x kernel (when no -rc kernels have been released yet) or to a Linus -rc kernel.

Here are some examples of applying the -mm patches:

```
# moving from 2.6.12 to 2.6.12-mm1
$ cd ~/linux-2.6.12                     # change to the 2.6.12 source dir
$ patch -p1 < ../2.6.12-mm1             # apply the 2.6.12-mm1 patch
$ cd ..
$ mv linux-2.6.12 linux-2.6.12-mm1      # rename the source appropriately

# moving from 2.6.12-mm1 to 2.6.13-rc3-mm3
$ cd ~/linux-2.6.12-mm1
$ patch -p1 -R < ../2.6.12-mm1          # revert the 2.6.12-mm1 patch
                                        # we now have a 2.6.12 source
$ patch -p1 < ../patch-2.6.13-rc3       # apply the 2.6.13-rc3 patch
                                        # we now have a 2.6.13-rc3 source
$ patch -p1 < ../2.6.13-rc3-mm3         # apply the 2.6.13-rc3-mm3 patch
$ cd ..
$ mv linux-2.6.12-mm1 linux-2.6.13-rc3-mm3      # rename the source dir
```

This concludes this list of explanations of the various kernel trees.
I hope you are now clear on how to apply the various patches and help testing

the kernel.

Thank you's to Randy Dunlap, Rolf Eike Beer, Linus Torvalds, Bodo Eggert, Johannes Stezenbach, Grant Coady, Pavel Machek and others that I may have forgotten for their reviews and contributions to this document.