FEATURED STORIES

- Why Solaris 10 for x86 is Worth A Look
- Build a Compute Cluster for $1,500
- Read Our Exclusive *Linux and the Law* Column
- Interview with Linus Torvalds
- Why China is Embracing Linux
- A Look at All Things Debian

MOST POPULAR STORIES

- It's All About the Debians
- The Emperor Penguin
- Interview: Branden Robinson
- Network Nirvana
- Securing your Environment, Part One
- Linux in the Red
- Say Hello to Skype
- jCIFS: The SMB Can Opener
- And Now, Something Completely Different
- It's (Not) Magic

MOST RECENT POSTS

- Talking

Journaling File Systems

Feature Story

Written by Steve Best

Tuesday, 15 October 2002

The file system is one of the most important parts of an operating system. The file system stores and manages user data on disk drives, and ensures that what's read from storage is identical to what was originally written. In addition to storing user data in files, the file system also creates and manages information about files and about itself. Besides guaranteeing the integrity of all that data, file systems are also expected to be extremely reliable and have very good performance.

For the past several years, Ext2 has been the de facto file system for most Linux machines. It's robust, reliable, and suitable for most deployments. However, as Linux displaces Unix and other operating systems in more and more large server and computing environments, Ext2 is being pushed to its limits. In fact, many now common requirements -- large hard-disk partitions, quick recovery from crashes, high-performance I/O, and the need to store thousands and thousands of files representing terabytes of data -- exceed the abilities of Ext2.

OpenOffice.org
- Michael Dell Betting On Red Hat
- History Of The Linux Kernel Archives
- March 2005 Now Online
- Red Hat Inks Deal With DoD
- Gael Duval Interview
- New Debian Servers

LINUX IS OPEN FOR BUSINESS

LINUX solutions

THIS SITE POWERED BY...

debian

Fortunately, a number of other Linux file systems take up where Ext2 leaves off. Indeed, Linux now offers four alternatives to Ext2: Ext3, ReiserFS, XFS, and JFS. In addition to meeting some or all of the requirements listed above, each of these alternative file systems also supports *journaling*, a feature certainly demanded by enterprises, but beneficial to anyone running Linux. A journaling file system can simplify restarts, reduce fragmentation, and accelerate I/O. Better yet, journaling file systems make *fsck*s a thing of the past.

If you maintain a system of fair complexity or require high-availability, you should seriously consider a journaling file system. Let's find out how journaling file systems work, look at the four journaling file systems available for Linux, and walk through the steps of installing one of the newer systems, JFS. Switching to a journaling file system is easier than you might think, and once you switch -- well, you'll be glad you did.

Fun with File Systems

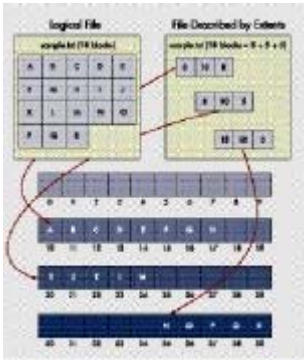
To better appreciate the benefits of journaling file systems, let's start by looking at how files are saved in a non-journaled file system like Ext2. To do that, it's helpful to speak the vernacular of file systems.

- A *logical block* is the smallest unit of storage that can be allocated by the file system. A logical block is measured in bytes, and it may take several blocks to store a single file.
- A *logical volume* can be a physical disk or some subset of the physical disk space. A logical volume is also known as a *disk partition*.
- *Block allocation* is a method of allocating blocks where the file system allocates one block at a time. In this method, a pointer to every block in a file is maintained and recorded.
- *Internal fragmentation* occurs when a file does not a fill a block completely. For example, if a file is 10K and a block is 8K, the file system allocates two blocks to hold the file, but 6K is wasted. Notice that as blocks get bigger, so does the potential to have waste.
- *External fragmentation* occurs when the logical blocks that make up a file are scattered all over the disk. External fragmentation can cause poor performance.
- An *extent* is a large number of contiguous blocks. Each extent is described by a triple, consisting of *(file offset, starting block number, length)*, where *file offset* is the offset of the extent's first block from the beginning of the file, *starting block number* is the first block in the extent, and *length* is the number of blocks in the extent. Extents are allocated and tracked as a single unit, meaning that a single pointer tracks a group of blocks. For large files, *extent allocation* is a much more

efficient technique than block allocation. *Figure One* shows how extents are used.

- File system *meta-data* is the file system's internal data structures -- everything concerning a file except the actual data inside the file. Meta-data includes date and time stamps, ownership information, file access permissions, other security information such as access control lists (if they exist), the file's size and the storage location or locations on disk.
- An *inode* stores all of the information *about* a file except the data itself. You can think of an inode as a "bookkeeping" file for a file (indeed, an inode is a file that consumes blocks, too). An inode contains file permissions, file types, and the number of links to the file. It can also contain some direct pointers to file data blocks; pointers to blocks that contain pointers to file data bocks (so-called indirect pointers); and even double- and triple-indirect pointers. Every inode has a unique inode number that distinguishes it from every other inode.
- A *directory* is a special kind of file that simply contains pointers to other files. Specifically, the inode for a directory file simply contains the inode numbers of its contents, plus permissions, etc.

Figure One: How file extents work



An extent is described by its block offset in the file, the location of the first block in the extent, and the length of the extent.

If file *sample.tx*t requires 18 blocks, and the file system is able to allocate one extent of length 8, a second extent of length 5, and a third extent of length 5, the file system would look something like the drawing below. The first extent has offset 0 (block Ain the file), location 0, and length 8. The second extent has offset 8 (block `I`), location 20, and length 5. The last extent has offset 13, location 35, and length 5.

Figure Two illustrates blocks, inodes (with a number of meta-data attributes), directories, and their relationships.

When Good File Systems Go Bad

With those concepts in mind,

here's what happens when a three-block file is modified and grows to be a five-block file:

- First, two new blocks are allocated to hold the new data.
- Next, the file's inode is updated to record the two new block pointers and the new size of the file.
- Finally, the actual data is written into the blocks.

Figure Two: Blocks, inodes, directories, files, and their relationships

As you can see, while writing data to a file appears to be a single atomic operation, the actual process involves a number of steps (even more steps than shown here if you consider all of the accounting required to remove free blocks from a list of free blocka, among other possible metadata changes).

If all the steps to write a file are completed perfectly (and this happens most of the time), the file is saved successfully. However, if the process is interrupted at any time (perhaps due to power failure or other systemic failure), a non-journaled file system can end up in an inconsistent state. Corruption occurs because the logical operation of writing (or updating) a file is actually a sequence of I/O, and the entire operation may not be totally reflected on the media at any given point in time.

If the meta-data or the file data is left in an inconsistent state, the file system will no longer function properly.

Non-journaled file systems rely on *fsck* to examine all of the file system's metadata and detect and repair structural integrity problems before restarting. If Linux shuts down smoothly, *fsck* will typically return a clean bill of health. However, after a power failure or crash, *fsck* is likely to find some kind of error in meta-data.

A file system has a lot of meta-data, and *fsck* can be very time consuming. After all, *fsck* has to scan a file system's *entire* repository of meta-data to ensure consistency and error-free operation. As you may have experienced, the speed of *fsck* on a disk partition is proportional to the size of the partition, the number of directories, and the number of files in each directory.

For large file systems, journaling becomes crucial. A journaling file system provides improved structural consistency, better recovery, and faster restart times than non-journaled file systems. In most cases, a journaled file system can restart in less than a second.

Dear Journal...

The magic of journaling file systems lies in *transactions*. Just like a database transaction, a journaling file system transaction treats a sequence of changes as a single, atomic operation -- but instead of tracking updates to tables, the journaling file system tracks changes to file system meta-data and/or user data. The transaction guarantees that either *all* or *none* of the file system updates are done.

For example, the process of creating a new file modifies several meta-data structures (inodes, free lists, directory entries, etc.). Before the file system makes those changes, it creates a transaction that describes what it's about to do. Once the transaction has been recorded (on disk), the file system goes ahead and modifies the meta-data. The *journal* in a journaling file system is simply a list of transactions.

In the event of a system failure, the file system is restored to a consistent state by replaying the journal. Rather than examine *all* meta-data (the *fsck* way), the file system inspects only those portions of the meta-data that have recently changed. Recovery is much faster, usually only a matter of seconds. Better yet, recovery time is not dependent on the size of the partition.

In addition to faster restart times, most journaling file systems also address another significant problem: scalability. If you combine even a few large-capacity disks, you can assemble some massive (certainly by early-90s' standards) file systems. Features of modern file systems include:

- Faster allocation of free blocks. Extents (as described above) and B+ trees are used individually or together to find and allocate several free blocks, either by size or location, quickly.
- Large (or very large) numbers of files in a directory. A directory is a special file that contains a list of files. If you want a directory to contain thousands or tens of thousands of files, something better than a linked-list of (name, inode) pairs is needed. Again, advanced file systems used B+ trees to store directory entries. In some cases, a single B+ tree is used for the entire system.
- Large files. The old technique of storing direct, indirect, double-indirect, and even triple indirect pointers to blocks does not scale well. For very large files, the number of disk accesses needed to retrieve a block in the data file would be prohibitively expensive.

More advanced file systems also manage sparse files, internal fragmentation, and the allocation of inodes better than Ext2.

A Wealth of Options

While advanced file systems are tailored primarily for the high throughput and high uptime requirements of servers

(from single processor systems to clusters), these file systems can also benefit client machines where performance and reliability are wanted or needed.

As mentioned in the introduction, recent releases of Linux include not one, but four journaling file systems. JFS from IBM, XFS from SGI, and ReiserFS from Namesys have all been "open sourced" and subsequently included in the Linux kernel. In addition, Ext3 was developed as a journaling add-on to Ext2.

Figure Three shows where the file systems fit in Linux. You'll note that JFS, XFS, ReiserFS, and Ext3 are independent "peers." It's possible for a single Linux machine to use all of those file systems at the same time. A system administrator could configure a system to use XFS on one partition, and ReiserFS on another.

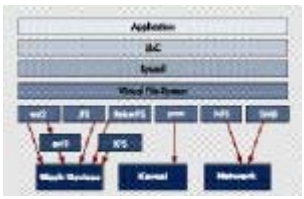


Figure Three: Where file systems fit in the operating system

What are the features and benefits of each system? Let's take a quick look at Ext3, ReiserFS, and XFS, and then an in-depth look at JFS.

EXT3

As mentioned above, Ext2 is the de facto file system for Linux. While it lacks some of the advanced features (extremely large files, extent-mapped files, etc.) of XFS and ReiserFS and others, it's reliable, stable, and still the default "out of the box" file system for all Linux distributions. Ext2's real weakness is *fsck*: the bigger the Ext2 file system, the longer it takes to *fsck*. Longer *fsck* times means longer down times.

The Ext3 file system was designed to provide higher availability without impacting the robustness (at least the simplicity and reliability) of Ext2. Ext3 is a minimal extension to Ext2 to add support for journaling. Ext3 uses the same disk layout and data structures as Ext2, and it's forward- and backward-compatible with Ext2. Migration from Ext2 to Ext3 (and vice versa) is quite easy, and can even be done in-place in the same partition. The other three journaling file systems required the partition to be formatted with their *mkfs* utility.

If you want to adopt a journaling file system, but don't have free partitions on your system, Ext3 could be the journaling file system to use. See "Switching to Ext3" for information on how to switch to Ext3 on your Linux machine.

Switching to Ext3

If you want to switch to Ext3, it's a good idea to make a backup of your file systems. Once you've done that, run the *tune2fs* program

with the `-j` option to add a journal file to an existing Ext2 file system. You can run *tune2fs* on a mounted or unmounted Ext2 file system. For instance, if */dev/hdb3* is an Ext2 file system, the command

```
# tune2fs -j /dev/hdb3
```

creates the log. If the file system is mounted, a journal file named *.journal* will be placed in the root directory of the file system. If the file system is not mounted, the journal file will be hidden. (When you mount an Ext3 file system, the *.journal* file will appear. The *.journal* file is just an indicator to show that the file system is indeed Ext3.)

Next, the entry for */dev/hdb* in */etc/fstab* needs to be changed from `ext2` to `ext3`. The final step is to reboot and verify that the */dev/hdb3* partition has type `ext3`. Type *mount*. The output should include an entry like this one:

```
% mount

/dev/hdb3 on /test type ext3 (rw)
```

Ext3 provides three data journaling modes that can be set at mount time: `data=journal`, `data=writeback`, and `data=ordered`. The `data=journal` mode provides both meta-data and data journaling. `data=writeback` mode provides only meta-data journaling. `data=ordered` mode, which is the default mode, provides meta-data journaling with increased integrity. With three modes, a system administrator can make a trade off between performance and file data consistency.

If for some reason you'd like to change the Ext3 partition back to Ext2, the process is very simple: *umount* the file system, and re-mount it using Ext2.

```
# mount -t ext2 /dev/hdb3 /test
```

If you want the file system to mount as Ext2 at boot time, you'll also have to change its entry in *etc/fstab*.

The downside of Ext3? It's an add-on to Ext2, so it still has the same limitations that Ext2 has. The fixed internal structures of Ext2 are simply too small (too few bits) to

capture large file sizes, extremely large partition sizes, and enormous numbers of files in a single directory. Moreover, the bookkeeping techniques of Ext2, such as its linked-list directory implementation, do not scale well to large file systems (there is an upper limit of 32,768 subdirectories in a single directory, and a "soft" upper limit of 10,000-15,000 files in a single directory.) To make radical improvements to Ext2, you'd have to make radical changes. Radical change was not the intent of Ext3.

However, newer file systems do not have to be backward-compatible with Ext2. ReiserFS, XFS, and JFS offer scalability, high-performance, very large file systems, and of course, journaling. "Why Four Journaling File Systems is a Good Thing" presents an overview of the capabilities of the four journaling file systems.

Why Four Journaling File Systems is Good

One of the great things about open source is that choice is looked upon favorably. Linux is the only operating system with four journaling file systems in production: ReiserFS, Ext3, JFS, and XFS.

All four file systems have the GPL license, and source code is available at http://www.kernel.org or on each project's home page. Each of the journaling file system teams follow a community model and welcome users and contributors. In fact, the teams share their best ideas, and competitive benchmarking encourages constant improvement of all of the systems.

The table below summarizes the features and limits of the four Linux journaling file systems. The first section provides some history of when the journaling file system were accepted into the kernel.org source trees. The next section, lists some of the features of the file systems. The final section, lists some of the distributions that are currently shipping the journaling file systems. If the distribution is shipping the file system that you want to use, you can use that file system right "out-of-the-box."

For complete feature lists of each journaling file system, see the respective project Web pages.

A comparison of journaling file systems

Kernel support	*Ext3*	*ReiserFS*	*XFS*	*JFS*
Kernel prerequisites	No	No	Yes	No
In kernel.org source tree 2.4.I*x*	2.4.15	2.4.1	-	-

In kernel.org source tree 2.5.I*x*	2.5.0	2.5.0	-	2.5.6
License	GPL	GPL	GPL	GPL
Features				
Largest block size supported on ia32	4 Kb	4 Kb	4 Kb	4 Kb
File system size maximum	16384 Gb	17592 Gb	18,000 Pb+	32 Pb
File size maximum	2048 Gb	1 Eb*	9,000 Pb	4 Pb
Growing the file system size	Patch	Yes	Yes	Yes
Access Control Lists	Patch	No	Yes	WIP
Dynamic disk inode allocation	No	Yes	Yes	Yes
Data logging	Yes	No	No	No
Place log on an external device	Yes	Yes	Yes	Yes
Distros with journaling file systems				
Red Hat 7.3	Yes	Yes	No	Yes
SuSE 8.0	Yes	Yes	Yes	Yes
Mandrake Linux 8.2	Yes	Yes	Yes	Yes
Slackware Linux 8.1	Yes	Yes	Yes	Yes

+ Pb is petabyte, or 10^{15} bytes

* Eb is exabyte or 10^{18} bytes

By the way, the 2.4 kernel has a limit of 2048 Gb for a single block device, so no file system larger than that can be created at this time (without patching the standard kernel). This restriction could be removed in the 2.5.x development kernel, and there are patches available to remove this limit, but as of 2.5.29, the patches haven't been officially included yet.

REISERFS

ReiserFS is designed and developed by Hans Reiser and his team of developers at Namesys. Like the other journaling file systems, it's open source, is available in most Linux distributions, and supports meta-data journaling.

One of the unique advantages of ReiserFS is support for *small* files -- lots and lots of small files. Reiser's philosophy is simple: small files encourage coding simplicity. Rather than use a database or create your own file caching scheme, use the filesystem to handle lots of small pieces of information.

ReiserFS is about eight to fifteen times faster than Ext2 at handling files smaller than 1K.

Even more impressive, (when properly configured) ReiserFS can actually store about 6% more data that Ext2 on the same physical file system. Rather than allocate space in fixed 4K blocks, ReiserFS can allocate the exact space that's needed. A B* tree manages all file system meta-data, *and* stores and compresses *tails*, portions of files smaller than a block.

Of course, ReiserFS also has excellent performance for large files, but it's especially adept at managing small files.

For a more in-depth discussion of ReiserFS and instructions on how to install it, see "Journaling File Systems" in the August 2000 issue, available online at http://www.linux-mag.com/2000-08/journaling_01.html.

JFS

JFS for Linux is based on IBM's successful JFS file system for OS/2 Warp. Donated to open source in early 2000 and ported to Linux soon after, JFS is well-suited to enterprise environments. JFS uses many advanced techniques to boost performance, provide for very large file systems, and of course, journal changes to the file system. SGI's XFS (described next) has many similar features. Some of the features of JFS include:

- Extent-based addressing structures. JFS uses extent-based addressing structures, along with aggressive block allocation policies to produce compact, efficient, and scalable structures for mapping logical offsets within files to physical addresses on disk. This feature yields excellent performance.
- Dynamic inode allocation. JFS dynamically allocates space for disk inodes as required, freeing the space when it is no longer required. This is a radical improvement over Ext2, which reserves a fixed amount of space for disk inodes at file system creation time. With dynamic inode allocation, users do not have to estimate the maximum number of files and directories that a file system will contain. Additionally, this feature decouples disk inodes from fixed disk locations.
- Directory organization. Two different directory organizations are provided: one is used for small directories and the other for large directories. The contents of a small directory (up to 8 entries, excluding the self (`.` or "dot") and parent (`..` or

"dot dot" entries) are stored within the directory's inode. This eliminates the need for separate directory block I/O and the need to allocate separate storage. The contents of larger directories are organized in a B+ tree keyed on name. B+ trees provide faster directory lookup, insertion, and deletion capabilities when compared to traditional unsorted directory organizations.

- 64-bits. JFS is a full 64-bit file system. All of the appropriate file system structure fields are 64-bits in size. This allows JFS to support large files and partitions.

There are other advanced features in JFS such as allocation groups (which speeds file access times by maximizing locality), and various block sizes ranging from 512-bytes to 4096-bytes (which can be tuned to avoid internal and external fragmentation). You can read about all of them at the JFS Web site at http://www-124.ibm.com/developerworks/oss/jfs.

XFS

A little more than a year ago, SGI released a version of its high-end XFS file system for Linux. Based on SGI's Irix XFS file system technology, XFS supports meta-data journaling, and extremely large disk farms. How large? A single XFS file system can be 18,000 *petabytes* (that's 10^{15} bytes) and a single *file* can be 9,000 petabytes. XFS is also capable of delivering excellent I/O performance.

In addition to truly amazing scale and speed, XFS uses many of the same techniques found in JFS.

Installing JFS

For the rest of the article, let's look at how to install and use IBM's JFS system. If you have the latest release of Turbolinux, Mandrake, SuSE, Red Hat, or Slackware, you can probably skip ahead to the section "Creating a JFS Partition." If you want to include the latest JFS source code drop into your kernel, the next few sections show you what to do.

THE LATEST AND GREATEST

JFS has been incorporated into the 2.5.6 Linux kernel, and is also included in Alan Cox's 2.4.*X*-ac kernels beginning with 2.4.18-pre9-ac4, which was released on February 14, 2002. Alan's patches for 2.4.*x* series are available from http://www.kernel.org. You can also download a 2.4 kernel source tree and add the JFS patches to this tree. JFS comes as a patch for several of the 2.4.*x* kernel, so first of all, get the latest kernel from http://www.kernel.org.

At the time of writing, the latest kernel was 2.4.18 and the latest release of JFS was 1.0.20. We'll be using those in the instructions below. The JFS patch is available from the JFS

web site. You also need both the utilities (*jfsutils-1.0.20.tar.gz*), the kernel patch (*jfs-2.4.18-patch*), and the file system source (*jfs-2.4-1.0.20.tar.gz*).

If you're using any of the latest distros, you probably won't have to patch the kernel for the JFS code. Instead, you'll only need to compile the kernel to update to the latest release of JFS (you can build JFS either as built-in or as a module). (To determine what version of JFS was shipped in the distribution you're running, you can edit the JFS file *super.c* and look for a `printk()` that has the JFS development version number string.)

PATCHING THE KERNEL TO SUPPORT JFS

In the example below, we'll use the 2.4.18 kernel source tree as an example on how to patch JFS into the kernel source tree.

First, you need to download the Linux kernel: linux-2.4.18 .tar.gz. If you have a *linux* subdirectory, move it to *linux-org*, so it won't replaced by the linux-2.4.18 source tree. When you download the kernel archive, save it under */usr/src* and expand the kernel source tree by using:

```
% mv linux linux-org
% tar zxvf linux-2.4.18.tar.gz
```

This operation will create a directory named */usr/src/linux*.

The next step is to get the JFS utilities and the appropriate patch for kernel 2.4.18. Before you do that, you need to create a directory for JFS source, */usr/src/jfs1020*, and download (to that directory) the JFS kernel patch and the JFS file system source files. Once you have those files, you have everything you need to patch the kernel.

Next, change to the directory of the kernel 2.4.18 source tree and apply the JFS kernel patch:

```
% cd /usr/src/linux
% patch -p1 < /usr/src/jfs1020/jfs-2.4-18-patch
% cp /usr/src/jfs1020/jfs-2.4-1.0.20.tar.gz .
% tar zxvf jfs-2.4-1.0.20.tar.gz
```

Now, you need to configure the kernel and enable JFS by going to the *File* systems section of the configuration menu and enabling *JFS file system support (*`CONFIG_JFS_FS=y`). You also have the option to configure JFS as a module, in which case you only need to recompile and reinstall kernel modules by typing:

```
% make modules && make install_modules
```

Otherwise, if you configured the JFS option as a kernel built-in, you need to:

1. Recompile the kernel (in */usr/src/linux*). Run the command

```
% make dep && make clean && make bzImage
```

2. Recompile and install modules (only if you added other options as modules)

```
% make modules && make modules_install
```

3. Install the kernel.

```
# cp arch/i386/boot/bzImage /boot/jfs-bzImage
# cp System.map /boot/jfs-System.map
# ln -s /boot/jfs-System.map /boot/System.map
```

Next, update */etc/lilo.conf* with the new kernel. Add an entry like the one that follows and a `jfs1020` entry should appear at the *lilo* boot prompt:

```
image=/boot/jfs-bzImage
label=jfs1020
read-only
root=/dev/hda5  # Change to your partition
```

Be sure to specify the correct root partition. Then run

```
# lilo
```

to make the system aware of the new kernel. Reboot and select the `jfs1020` kernel to boot from the new image.

After you compile and install the kernel, you should compile and install the JFS utilities. Save the *jfsutils-1.0.20.tar.gz* file into the */usr/src/jfs1020* directory, expand it, run *configure*, and the install the utilities.

```
% tar zxvf jfsutils-1.0.20.tar.gz
% cd jfsutils-1.0.20
% ./configure
% make && make install
```

Creating a JFS partition

Having built and installed the JFS utilities, the next step is to create a JFS partition. In this exact example, we'll demonstrate the process using a spare partition.

(If there's unpartitioned space on your disk, you can create a partition using *fdisk*. After you create the partition, reboot the system to make sure that the new partition is available to create a JFS file system on it. In our test system, we had */dev/hdb3* as a spare partition.)

To create the JFS file system with the log inside the JFS partition, apply the following command:

```
# mkfs.jfs /dev/hdb3
```

After the file system has been created, you need to mount it. You will need a mount point. Create a new empty directory such as */jfs* to mount the file system with the following

command:

```
# mount -t jfs /dev/hdb3 /jfs
```

After the file system is mounted, you are ready to try out JFS. To unmount the JFS file system, you simply use the *umount* command with the same mount point as the argument:

```
# umount /jfs
```

A Performance Tweak for All File Systems

Linux records an *atime*, or access time, whenever a file is read. However, access time isn't very useful, and can be quite costly to track.

To get a quick performance boost on any kind of Linux file system, simply disable access time updates with the *mount* option `noatime`. For example, to disable access times on a JFS partition, do something like this in */etc/fstab*:

```
/dev/hda6 /jfs jfs noatime 1 2
```

Go Faster with An External Log

An external log improves performance since the log updates are saved to a different partition than its corresponding file system.

To create the JFS file system with the log on an external device, your system will need to have 2 unused partitions. Our test system had */dev/hda6* and */dev/hdb1* as spare partitions.

```
# mkfs.jfs -j /dev/hdb1 /dev/hda6
mkfs.jfs version: 1.0.20 21-Jun-2002
Warning! All data on device /dev/hda6 will be lost!
Warning! All data on device /dev/hdb1 will be lost!
Continue? (Y/N) y
Format completed successfully.
10249438 kilobytes total disk space.
```

To mount the file system use the following mount command:

```
# mount -t jfs /dev/hda6 /jfs
```

So you don't have to mount this file system every time you boot, you can add it to */etc/fstab*. Make a backup of */etc/fstab* and edit it with you favorite editor. Add the */dev/hda6* device. For example, add:

```
/dev/hda6 /jfs jfs defaults 1 2
```

Not Just for Reboots Anymore

Some people have the impression that journaling file systems only provide fast restart times. As you've seen, this isn't true. Considerable coding efforts have made journaling file systems scalable, reliable, and fast.

Whether you're running an enterprise server, a cluster supercomputer, or a small Web site, XFS, JFS, and ReiserFS add credibility and oomph to Linux. Need a better reason to switch to a journaling file system? Just imagine yourself in a world without *fsck*. What will you do with all that extra time?

Resources

Ext3:
http://www.zipworld.com.au/~akpm/linux/ext3

JFS for Linux: http://oss.software.ibm.com/jfs

ReiserFS: http://www.namesys.com

Linux XFS: http://oss.sgi.com/projects/xfs

Extended Attributes & Access Controls Lists:
http://acl.bestbits.at

Steve Best works in the Linux Technology Center of IBM in Austin, Texas. He is currently working on the Journaled File System (JFS) for Linux project. Steve has done extensive work in operating system development with a focus in the areas of file systems, internationalization, and security. He can be reached at sbest@us.ibm .