The Linux NTFS filesystem driver

Overview

Linux-NTFS comes with a number of user-space programs known as ntfsprogs. These include mkntfs, a full-featured ntfs filesystem format utility, ntfsundelete used for recovering files that were unintentionally deleted from an NTFS volume and ntfsresize which is used to resize an NTFS partition. See the web site for more information.

To mount an NTFS 1.2/3.x (Windows NT4/2000/XP/2003) volume, use the file system type 'ntfs'. The driver currently supports read-only mode (with no fault-tolerance, encryption or journalling) and very limited, but safe, write support.

For fault tolerance and raid support (i.e. volume and stripe sets), you can use the kernel's Software RAID / MD driver. See section "Using Software RAID with NTFS" for details.

Web site

There is plenty of additional information on the linux-ntfs web site at http://www.linux-ntfs.org/

The web site has a lot of additional information, such as a comprehensive FAQ, documentation on the NTFS on-disk format, information on the Linux-NTFS userspace utilities, etc.

Features

- This is a complete rewrite of the NTFS driver that used to be in the 2.4 and earlier kernels. This new driver implements NTFS read support and is functionally equivalent to the old ntfs driver and it also implements limited write support. The biggest limitation at present is that files/directories cannot be created or deleted. See below for the list of write features that are so far supported. Another limitation is that writing to compressed files is not implemented at all. Also, neither read nor write access to encrypted files is so far implemented.
- The new driver has full support for sparse files on NTFS 3.x volumes which the old driver isn't happy with.
- The new driver supports execution of binaries due to mmap() now being supported.
- The new driver supports loopback mounting of files on NTFS which is used by some Linux distributions to enable the user to run Linux from an NTFS partition by creating a large file while in Windows and then loopback mounting the file while in Linux and creating a Linux filesystem on it that is used to install Linux on it.
- A comparison of the two drivers using:

  ```
  time find . -type f -exec md5sum "{}" \;
  ```

  run three times in sequence with each driver (after a reboot) on a 1.4GiB NTFS partition, showed the new driver to be 20% faster in total time elapsed (from 9:43 minutes on average down to 7:53). The time spent in user space was unchanged but the time spent in the kernel was decreased by a factor of 2.5 (from 85 CPU seconds down to 33).
- The driver does not support short file names in general. For backwards compatibility, we implement access to files using their short file names if they exist. The driver will not create short file names however, and a rename will discard any existing short file name.
- The new driver supports exporting of mounted NTFS volumes via NFS.
- The new driver supports async io (aio).
- The new driver supports fsync(2), fdatasync(2), and msync(2).
- The new driver supports readv(2) and writev(2).
- The new driver supports access time updates (including mtime and ctime).
- The new driver supports truncate(2) and open(2) with O_TRUNC. But at present only very limited support for highly fragmented files, i.e. ones which have their data attribute split across multiple extents, is included. Another limitation is that at present truncate(2) will never create sparse files, since to mark a file sparse we need to modify the directory entry for the file and we do not implement directory modifications yet.
- The new driver supports write(2) which can both overwrite existing data and extend the file size so that you can write beyond the existing data. Also, writing into sparse regions is supported and the holes are filled in with clusters. But at present only limited support for highly fragmented files, i.e. ones which have their data attribute split across multiple extents, is included. Another limitation is that write(2) will never create sparse files, since to mark a file sparse we need to modify the directory entry for the file and we do not implement directory modifications yet.

Supported mount options

In addition to the generic mount options described by the manual page for the mount command (man 8 mount, also see man 5 fstab), the NTFS driver supports the following mount options:

<table>
<tr><td>iocharset=name</td><td>Deprecated option. Still supported but please use nls=name in the future. See description for nls=name.</td></tr>
<tr><td>nls=name</td><td>Character set to use when returning file names. Unlike VFAT, NTFS suppresses names that contain unconvertible characters. Note that most character sets contain insufficient characters to represent all possible Unicode characters that can exist on NTFS. To be sure you are not missing any files, you are advised to use nls=utf8 which is capable of representing all Unicode characters.</td></tr>
<tr><td>utf8=<bool></td><td>Option no longer supported. Currently mapped to nls=utf8 but please use nls=utf8 in the future and make sure utf8 is compiled either as module or into the kernel. See description for nls=name.</td></tr>
<tr><td>uid=</td><td></td></tr>
<tr><td>gid=</td><td></td></tr>
<tr><td>umask=</td><td>Provide default owner, group, and access mode mask. These options work as documented in mount(8). By default, the files/directories are owned by root and he/she has read and write permissions, as well as browse permission for directories. No one else has any access permissions. I.e. the mode on all files is by default rw------- and for directories rwx------, a consequence of the default fmask=0177 and dmask=0077. Using a umask of zero will grant all permissions to everyone, i.e. all files and directories will have mode rwxrwxrwx.</td></tr>
<tr><td>fmask=</td><td></td></tr>
<tr><td>dmask=</td><td>Instead of specifying umask which applies both to files and directories, fmask applies only to files and dmask only to directories.</td></tr>
<tr><td>sloppy=<BOOL></td><td>If sloppy is specified, ignore unknown mount options. Otherwise the default behaviour is to abort mount if any unknown options are found.</td></tr>
<tr><td>show_sys_files=<BOOL></td><td>If show_sys_files is specified, show the system files in directory listings. Otherwise the default behaviour is to hide the system files. Note that even when show_sys_files is specified, "$MFT" will not be visible due to bugs/mis-features in glibc. Further, note that irrespective of show_sys_files, all files are accessible by name, i.e. you can always do "ls -l $UpCase" for example to specifically show the system file containing the Unicode upcase table.</td></tr>
<tr><td>case_sensitive=<BOOL></td><td>If case_sensitive is specified, treat all file names as case sensitive and create file names in the POSIX namespace. Otherwise the default behaviour is to treat file names as case insensitive and to create file names in the WIN32/LONG name space. Note, the Linux NTFS driver will never create short file names and will remove them on rename/delete of the corresponding long file name. Note that files remain accessible via their short file name, if it exists. If case_sensitive, you will need to provide the correct case of the short file name.</td></tr>
<tr><td>disable_sparse=<BOOL></td><td>If disable_sparse is specified, creation of sparse regions, i.e. holes, inside files is disabled for the volume (for the duration of this mount only). By default, creation of sparse regions is enabled, which is consistent with the behaviour of traditional Unix filesystems.</td></tr>
<tr><td>errors=opt</td><td>What to do when critical filesystem errors are found. Following values can be used for "opt":

<table><tr><td>continue</td><td>DEFAULT, try to clean-up as much as possible, e.g. marking a corrupt inode as bad so it is no longer accessed, and then continue.</td></tr><tr><td>recover</td><td>At present only supported is recovery of the boot sector from the backup copy. If read-only mount, the recovery is done in memory only and not written to disk.</td></tr></table>
Note that the options are additive, i.e. specifying:

<pre>errors=continue,errors=recover</pre>
means the driver will attempt to recover and if that fails it will clean-up as much as possible and continue.</td></tr>
</table>

<table>
<tr><td>mft_zone_multiplier=</td><td>Set the MFT zone multiplier for the volume (this setting is not persistent across mounts and can be changed from mount to mount but cannot be changed on remount). Values of 1 to 4 are allowed, 1 being the default. The MFT zone multiplier determines how much space is reserved for the MFT on the volume. If all other space is used up, then the MFT zone will be shrunk dynamically, so this has no impact on the amount of free space. However, it can have an impact on performance by affecting fragmentation of the MFT. In general use the default. If you have a lot of small files then use a higher value. The values have the following meaning:

<table>
<tr><th>Value</th><th>MFT zone size (% of volume size)</th></tr>
<tr><td>1</td><td>12.5%</td></tr>
<tr><td>2</td><td>25%</td></tr>
<tr><td>3</td><td>37.5%</td></tr>
<tr><td>4</td><td>50%</td></tr>
</table>

Note this option is irrelevant for read-only mounts.</td></tr>
</table>

Known bugs and (mis-)features

- The link count on each directory inode entry is set to 1, due to Linux not supporting directory hard links. This may well confuse some user space applications, since the directory names will have the same inode numbers. This also speeds up ntfs_read_inode() immensely. And we haven't found any problems with this approach so far. If you find a problem with this, please let us know.

Please send bug reports/comments/feedback/abuse to the Linux-NTFS development list at sourceforge: linux-ntfs-dev@lists.sourceforge.net

Using NTFS volume and stripe sets

For support of volume and stripe sets, you can either use the kernel's Device-Mapper driver or the kernel's Software RAID / MD driver. The former is the recommended one to use for linear raid. But the latter is required for raid level 5. For striping and mirroring, either driver should work fine.

The Device-Mapper driver

You will need to create a table of the components of the volume/stripe set and how they fit together and load this into the kernel using the dmsetup utility (see man 8 dmsetup).

Linear volume sets, i.e. linear raid, has been tested and works fine. Even though untested, there is no reason why stripe sets, i.e. raid level 0, and mirrors, i.e. raid level 1 should not work, too. Stripes with parity, i.e. raid level 5, unfortunately cannot work yet because the current version of the Device-Mapper driver does not support raid level 5. You may be able to use the Software RAID / MD driver for raid level 5, see the next section for details.

To create the table describing your volume you will need to know each of its components and their sizes in sectors, i.e. multiples of 512-byte blocks.

For NT4 fault tolerant volumes you can obtain the sizes using fdisk. So for example if one of your partitions is /dev/hda2 you would do:

```
$ fdisk -ul /dev/hda

Disk /dev/hda: 81.9 GB, 81964302336 bytes
255 heads, 63 sectors/track, 9964 cylinders, total 160086528 sectors
Units = sectors of 1 * 512 = 512 bytes

    Device Boot      Start         End      Blocks   Id  System
    /dev/hda1   *          63     4209029     2104483+  83  Linux
    /dev/hda2         4209030    37768814    16779892+  86  NTFS
    /dev/hda3        37768815    46170809     4200997+  83  Linux
```

And you would know that /dev/hda2 has a size of 37768814 - 4209030 + 1 = 33559785 sectors.

For Win2k and later dynamic disks, you can for example use the ldminfo utility which is part of the Linux LDM tools (the latest version at the time of writing is linux-ldm-0.0.8.tar.bz2). You can download it from:

http://www.linux-ntfs.org/

Simply extract the downloaded archive (tar xvjf linux-ldm-0.0.8.tar.bz2), go into it (cd linux-ldm-0.0.8) and change to the test directory (cd test). You will find the precompiled (i386) ldminfo utility there. NOTE: You will not be able to compile this yourself easily so use the binary version!

Then you would use ldminfo in dump mode to obtain the necessary information:

```
$ ./ldminfo --dump /dev/hda
```

This would dump the LDM database found on /dev/hda which describes all of your dynamic disks and all the volumes on them. At the bottom you will see the VOLUME DEFINITIONS section which is all you really need. You may need to look further above to determine which of the disks in the volume definitions is which device in Linux. Hint: Run ldminfo on each of your dynamic disks and look at the Disk Id close to the top of the output for each (the PRIVATE HEADER section). You can then find these Disk Ids in the VBLK DATABASE section in the <Disk> components where you will get the LDM Name for the disk that is found in the VOLUME DEFINITIONS section.

Note you will also need to enable the LDM driver in the Linux kernel. If your distribution did not enable it, you will need to recompile the kernel with it enabled. This will create the LDM partitions on each device at boot time. You would then use those devices (for /dev/hda they would be /dev/hda1, 2, 3, etc) in the Device-Mapper table.

You can also bypass using the LDM driver by using the main device (e.g. /dev/hda) and then using the offsets of the LDM partitions into this device as the "Start sector of device" when creating the table. Once again ldminfo would give you the correct information to do this.

Assuming you know all your devices and their sizes things are easy.

For a linear raid the table would look like this (note all values are in 512-byte sectors):

```
# Offset into       Size of this    Raid type       Device          Start sector
# volume    device                                              of device
0           1028161         linear          /dev/hda1       0
1028161             3903762         linear          /dev/hdb2       0
4931923             2103211         linear          /dev/hdc1       0
```

For a striped volume, i.e. raid level 0, you will need to know the chunk size you used when creating the volume. Windows uses 64kiB as the default, so it will probably be this unless you changes the defaults when creating the array.

For a raid level 0 the table would look like this (note all values are in 512-byte sectors):

```
# Offset   Size         Raid     Number   Chunk  1st        Start   2nd        Start
# into     of the   type     of           size   Device     in      Device     in
# volume   volume        stripes                    device             device
0      2056320  striped  2        128    /dev/hda1  0       /dev/hdb1 0
```

If there are more than two devices, just add each of them to the end of the line.

Finally, for a mirrored volume, i.e. raid level 1, the table would look like this (note all values are in 512-byte sectors):

```
# Ofs Size   Raid   Log  Number Region Should Number Source  Start Target Start
# in  of the type   type of log size   sync?  of     Device  in    Device in
# vol volume                 params             mirrors         Device       Device
0    2056320 mirror core 2  16     nosync 2    /dev/hda1 0   /dev/hdb1 0
```

If you are mirroring to multiple devices you can specify further targets at the end of the line.

Note the "Should sync?" parameter "nosync" means that the two mirrors are already in sync which will be the case on a clean shutdown of Windows. If the mirrors are not clean, you can specify the "sync" option instead of "nosync" and the Device-Mapper driver will then copy the entirety of the "Source Device" to the "Target Device" or if you specified multiple target devices to all of them.

Once you have your table, save it in a file somewhere (e.g. /etc/ntfsvolume1), and hand it over to dmsetup to work with, like so:

```
$ dmsetup create myvolume1 /etc/ntfsvolume1
```

You can obviously replace "myvolume1" with whatever name you like.

If it all worked, you will now have the device /dev/device-mapper/myvolume1 which you can then just use as an argument to the mount command as usual to mount the ntfs volume. For example:

```
$ mount -t ntfs -o ro /dev/device-mapper/myvolume1 /mnt/myvol1
```

(You need to create the directory /mnt/myvol1 first and of course you can use anything you like instead of /mnt/myvol1 as long as it is an existing directory.)

It is advisable to do the mount read-only to see if the volume has been setup correctly to avoid the possibility of causing damage to the data on the ntfs volume.

The Software RAID / MD driver

An alternative to using the Device-Mapper driver is to use the kernel's Software RAID / MD driver. For which you need to set up your /etc/raidtab appropriately (see man 5 raidtab).

Linear volume sets, i.e. linear raid, as well as stripe sets, i.e. raid level 0, have been tested and work fine (though see section "Limitations when using the MD driver with NTFS volumes" especially if you want to use linear raid). Even though untested, there is no reason why mirrors, i.e. raid level 1, and stripes with parity, i.e. raid level 5, should not work, too.

You have to use the "persistent-superblock 0" option for each raid-disk in the NTFS volume/stripe you are configuring in /etc/raidtab

as the persistent superblock used by the MD driver would damage the NTFS volume.

Windows by default uses a stripe chunk size of 64k, so you probably want the "chunk-size 64k" option for each raid-disk, too.

For example, if you have a stripe set consisting of two partitions /dev/hda5 and /dev/hdb1 your /etc/raidtab would look like this:

```
raiddev /dev/md0
        raid-level  0
        nr-raid-disks       2
        nr-spare-disks      0
        persistent-superblock       0
        chunk-size  64k
        device              /dev/hda5
        raid-disk   0
        device              /dev/hdb1
        raid-disk   1
```

For linear raid, just change the raid-level above to "raid-level linear", for mirrors, change it to "raid-level 1", and for stripe sets with parity, change it to "raid-level 5".

Note for stripe sets with parity you will also need to tell the MD driver which parity algorithm to use by specifying the option "parity-algorithm which", where you need to replace "which" with the name of the algorithm to use (see man 5 raidtab for available algorithms) and you will have to try the different available algorithms until you find one that works. Make sure you are working read-only when playing with this as you may damage your data otherwise. If you find which algorithm works please let us know (email the linux-ntfs developers list linux-ntfs-dev@lists.sourceforge.net or drop in on IRC in channel #ntfs on the irc.freenode.net network) so we can update this documentation.

Once the raidtab is setup, run for example raid0run -a to start all devices or raid0run /dev/md0 to start a particular md device, in this case /dev/md0.

Then just use the mount command as usual to mount the ntfs volume using for example:

```
mount -t ntfs -o ro /dev/md0 /mnt/myntfsvolume
```

It is advisable to do the mount read-only to see if the md volume has been setup correctly to avoid the possibility of causing damage to the data on the ntfs volume.

Limitations when using the Software RAID / MD driver

Using the md driver will not work properly if any of your NTFS partitions have an odd number of sectors. This is especially important for linear raid as all data after the first partition with an odd number of sectors will be offset by one or more sectors so if you mount such a partition with write support you will cause massive damage to the data on the volume which will only become apparent when you try to use the volume again under Windows.

So when using linear raid, make sure that all your partitions have an even number of sectors BEFORE attempting to use it. You have been warned!

Even better is to simply use the Device-Mapper for linear raid and then you do not have this problem with odd numbers of sectors.