Direct Access for files

Motivation

The page cache is usually used to buffer reads and writes to files. It is also used to provide the pages which are mapped into userspace by a call to mmap.

For block devices that are memory-like, the page cache pages would be unnecessary copies of the original storage. The *DAX* code removes the extra copy by performing reads and writes directly to the storage device. For file mappings, the storage device is mapped directly into userspace.

Usage

If you have a block device which supports *DAX*, you can make a filesystem on it as usual. The *DAX* code currently only supports files with a block size equal to your kernel's *PAGE_SIZE*, so you may need to specify a block size when creating the filesystem.

Currently 5 filesystems support *DAX*: ext2, ext4, xfs, virtiofs and erofs. Enabling *DAX* on them is different.

Enabling DAX on ext2 and erofs

When mounting the filesystem, use the `-o dax` option on the command line or add 'dax' to the options in `/etc/fstab`. This works to enable *DAX* on all files within the filesystem. It is equivalent to the `-o dax=always` behavior below.

Enabling DAX on xfs and ext4

Summary

1. There exists an in-kernel file access mode flag *S_DAX* that corresponds to the statx flag *STATX_ATTR_DAX*. See the manpage for statx(2) for details about this access mode.
2. There exists a persistent flag *FS_XFLAG_DAX* that can be applied to regular files and directories. This advisory flag can be set or cleared at any time, but doing so does not immediately affect the *S_DAX* state.
3. If the persistent *FS_XFLAG_DAX* flag is set on a directory, this flag will be inherited by all regular files and subdirectories that are subsequently created in this directory. Files and subdirectories that exist at the time this flag is set or cleared on the parent directory are not modified by this modification of the parent directory.
4. There exist dax mount options which can override *FS_XFLAG_DAX* in the setting of the *S_DAX* flag. Given underlying storage which supports *DAX* the following hold:

   `-o dax=inode` means "follow *FS_XFLAG_DAX*" and is the default.

   `-o dax=never` means "never set *S_DAX*, ignore *FS_XFLAG_DAX*."

   `-o dax=always` means "always set *S_DAX* ignore *FS_XFLAG_DAX*."

   `-o dax` is a legacy option which is an alias for `dax=always`.

   > **Warning**
   >
   > The option `-o dax` may be removed in the future so `-o dax=always` is the preferred method for specifying this behavior.

   > **Note**
   >
   > Modifications to and the inheritance behavior of *FS_XFLAG_DAX* remain the same even when the filesystem is mounted with a dax option. However, in-core inode state (*S_DAX*) will be overridden until the filesystem is remounted with dax=inode and the inode is evicted from kernel memory.

5. The *S_DAX* policy can be changed via:
   a. Setting the parent directory *FS_XFLAG_DAX* as needed before files are created
   b. Setting the appropriate dax="foo" mount option
   c. Changing the *FS_XFLAG_DAX* flag on existing regular files and directories. This has runtime constraints and limitations that are described in 6) below.
6. When changing the *S_DAX* policy via toggling the persistent *FS_XFLAG_DAX* flag, the change to existing

regular files won't take effect until the files are closed by all processes.

Details

There are 2 per-file dax flags. One is a persistent inode setting (*FS_XFLAG_DAX*) and the other is a volatile flag indicating the active state of the feature (*S_DAX*).

FS_XFLAG_DAX is preserved within the filesystem. This persistent config setting can be set, cleared and/or queried using the *FS_IOC_FS`[`GS]`ETXATTR` ioctl (see ioctl_xfs_fsgetxattr(2)) or an utility such as 'xfs_io'.

New files and directories automatically inherit *FS_XFLAG_DAX* from their parent directory **when created**. Therefore, setting *FS_XFLAG_DAX* at directory creation time can be used to set a default behavior for an entire sub-tree.

To clarify inheritance, here are 3 examples:

Example A:

```
mkdir -p a/b/c
xfs_io -c 'chattr +x' a
mkdir a/b/c/d
mkdir a/e

------[outcome]------

dax: a,e
no dax: b,c,d
```

Example B:

```
mkdir a
xfs_io -c 'chattr +x' a
mkdir -p a/b/c/d

------[outcome]------

dax: a,b,c,d
no dax:
```

Example C:

```
mkdir -p a/b/c
xfs_io -c 'chattr +x' c
mkdir a/b/c/d

------[outcome]------

dax: c,d
no dax: a,b
```

The current enabled state (*S_DAX*) is set when a file inode is instantiated in memory by the kernel. It is set based on the underlying media support, the value of *FS_XFLAG_DAX* and the filesystem's dax mount option.

statx can be used to query *S_DAX*.

> **Note**
>
> That only regular files will ever have *S_DAX* set and therefore statx will never indicate that *S_DAX* is set on directories.

Setting the *FS_XFLAG_DAX* flag (specifically or through inheritance) occurs even if the underlying media does not support dax and/or the filesystem is overridden with a mount option.

Enabling DAX on virtiofs

The semantic of DAX on virtiofs is basically equal to that on ext4 and xfs, except that when '-o dax=inode' is specified, virtiofs client derives the hint whether DAX shall be enabled or not from virtiofs server through FUSE protocol, rather than the persistent *FS_XFLAG_DAX* flag. That is, whether DAX shall be enabled or not is completely determined by virtiofs server, while virtiofs server itself may deploy various algorithm making this decision, e.g. depending on the persistent *FS_XFLAG_DAX* flag on the host.

It is still supported to set or clear persistent *FS_XFLAG_DAX* flag inside guest, but it is not guaranteed that DAX will be enabled or disabled for corresponding file then. Users inside guest still need to call statx(2) and check the statx flag *STATX_ATTR_DAX* to see if DAX is enabled for this file.

Implementation Tips for Block Driver Writers

To support *DAX* in your block driver, implement the 'direct_access' block device operation. It is used to translate the sector number (expressed in units of 512-byte sectors) to a page frame number (pfn) that identifies the physical page for the memory. It also returns a kernel virtual address that can be used to access the memory.

The direct_access method takes a 'size' parameter that indicates the number of bytes being requested. The function should return the number of bytes that can be contiguously accessed at that offset. It may also return a negative errno if an error occurs.

In order to support this method, the storage must be byte-accessible by the CPU at all times. If your device uses paging techniques to expose a large amount of memory through a smaller window, then you cannot implement direct_access. Equally, if your device can occasionally stall the CPU for an extended period, you should also not attempt to implement direct_access.

These block devices may be used for inspiration: - brd: RAM backed block device driver - dcssblk: s390 dcss block device driver - pmem: NVDIMM persistent memory driver

Implementation Tips for Filesystem Writers

Filesystem support consists of:

- Adding support to mark inodes as being *DAX* by setting the *S_DAX* flag in i_flags
- Implementing ->read_iter and ->write_iter operations which use :c:func:`dax_iomap_rw()` when inode has *S_DAX* flag set

  System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]dax.rst`, **line 221**); ***backlink***

  Unknown interpreted text role "c:func".

- Implementing an mmap file operation for *DAX* files which sets the *VM_MIXEDMAP* and *VM_HUGEPAGE* flags on the *VMA*, and setting the vm_ops to include handlers for fault, pmd_fault, page_mkwrite, pfn_mkwrite. These handlers should probably call :c:func:`dax_iomap_fault()` passing the appropriate fault size and iomap operations.

  System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]dax.rst`, **line 223**); ***backlink***

  Unknown interpreted text role "c:func".

- Calling :c:func:`iomap_zero_range()` passing appropriate iomap operations instead of :c:func:`block_truncate_page()` for *DAX* files

  System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]dax.rst`, **line 228**); ***backlink***

  Unknown interpreted text role "c:func".

  System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]dax.rst`, **line 228**); ***backlink***

  Unknown interpreted text role "c:func".

- Ensuring that there is sufficient locking between reads, writes, truncates and page faults

The iomap handlers for allocating blocks must make sure that allocated blocks are zeroed out and converted to written extents before being returned to avoid exposure of uninitialized data through mmap.

These filesystems may be used for inspiration:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]dax.rst`, **line 239**)

Unknown directive type "seealso".

```
.. seealso::

  ext2: see Documentation/filesystems/ext2.rst
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]dax.rst, line 243)

Unknown directive type "seealso".

```
.. seealso::

  xfs:  see Documentation/admin-guide/xfs.rst
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]dax.rst, line 247)

Unknown directive type "seealso".

```
.. seealso::

  ext4: see Documentation/filesystems/ext4/
```

Handling Media Errors

The libnvdimm subsystem stores a record of known media error locations for each pmem block device (in gendisk->badblocks). If we fault at such location, or one with a latent error not yet discovered, the application can expect to receive a *SIGBUS*. Libnvdimm also allows clearing of these errors by simply writing the affected sectors (through the pmem driver, and if the underlying NVDIMM supports the clear_poison DSM defined by ACPI).

Since *DAX* IO normally doesn't go through the `driver/bio` path, applications or sysadmins have an option to restore the lost data from a prior `backup/inbuilt` redundancy in the following ways:

1. Delete the affected file, and restore from a backup (sysadmin route): This will free the filesystem blocks that were being used by the file, and the next time they're allocated, they will be zeroed first, which happens through the driver, and will clear bad sectors.
2. Truncate or hole-punch the part of the file that has a bad-block (at least an entire aligned sector has to be hole-punched, but not necessarily an entire filesystem block).

These are the two basic paths that allow *DAX* filesystems to continue operating in the presence of media errors. More robust error recovery mechanisms can be built on top of this in the future, for example, involving redundancy/mirroring provided at the block layer through DM, or additionally, at the filesystem level. These would have to rely on the above two tenets, that error clearing can happen either by sending an IO through the driver, or zeroing (also through the driver).

Shortcomings

Even if the kernel or its modules are stored on a filesystem that supports *DAX* on a block device that supports *DAX*, they will still be copied into RAM.

The DAX code does not work correctly on architectures which have virtually mapped caches such as ARM, MIPS and SPARC.

Calling :c:func:`get_user_pages()` on a range of user memory that has been mmaped from a *DAX* file will fail when there are no 'struct page' to describe those pages. This problem has been addressed in some device drivers by adding optional struct page support for pages under the control of the driver (see *CONFIG_NVDIMM_PFN* in `drivers/nvdimm` for an example of how to do this). In the non struct page cases *O_DIRECT* reads/writes to those memory ranges from a non-*DAX* file will fail

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]dax.rst, line 293); *backlink*

Unknown interpreted text role "c:func".

Note

O_DIRECT reads/writes _of a *DAX* file do work, it is the memory that is being accessed that is key here). Other things that will not work in the non struct page case include RDMA, :c:func:`sendfile()` and :c:func:`splice()`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]dax.rst, line 304); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\filesystems\[linux-master][Documentation][filesystems]dax.rst, line 304); *backlink*

Unknown interpreted text role "c:func".