Improving Block Discard Support throughout the Linux Storage Stack

Christoph Hellwig
What the heck are discards? - A very brief history of block I/O

- The traditional block interface simply was reads and writes of blocks.
- That’s nice and good for disks.
  - Well sorta..
- But Flash SSDs can not just overwrite existing data
  - So they must write out of place
  - And manage a block mapping
- Also enter under provisioned Arrays into the game
What the heck are discards? - A very brief history of block I/O

- We need a way to tell the device blocks aren’t in use anymore..
  - Linux calls this a discard
  - Every storage protocol has a different name for it
Different implementations of the discard concept: ATA TRIM

- ATA supports the *TRIM* operation in the *DSM* command
  - Supports up to 64 ranges
  - 16 bits worth of blocks per range
  - The *DSM* command is not queued

- Newer versions support queued TRIM
  - I’ve not actually seen a working implementation in the field
Different implementations of the discard concept: SCSI UNMAP

- SBC supports the **UNMAP** command
  - Supports an implementation specific number of ranges
  - 32 bits worth of blocks per range
  - All SCSI commands are queued
Different implementations of the discard concept: SCSI WRITE SAME

- SBC supports the WRITE SAME 10/16/32 commands to write a LBA sized buffer to many LBAs
  - If the UNMAP bit is set WRITE SAME ask the device to unmap the blocks covered
  - Buffer must be all zeros for the UNMAP bit to work.
  - Future reads from the LBAs must return all zeros
Different implementations of the discard concept: NVMe Deallocate

- NVMe supports the *Deallocate* operation in the **DSM** command
  - Supports up to 256 ranges, 32 bits worth of blocks per range
  - All NVMe commands are queued
When does the OS issue a discard?

1. Explicit through an ioctl:
   • e.g. mkfs time - trivial

2. Walk the free space information and discard everything that isn’t used:
   • (FITRIM ioctl, or horrible hacks in hdparm)

3. Whenever the file system actually frees previously space:
   • online discard (mount -o discard)
History of discard in Linux

- Support for **REQ_DISCARD** added in Linux 2.6.28 (2008):
  - Intended as a pure hint
  - Discards are issued asynchronously as “barriers”
  - Only single ranges supported
  - No payload in the bio / request
  - Exposed as **BLKDISCARD** ioctl
  - fat and ext4 support limited online discard
  - Implemented by MTD (raw flash)
History of discard in Linux (2)

- SCSI and ATA support added in Linux 2.6.33 (2009):
  - libata parses a SCSI WRITE SAME and translates it to an ATA TRIM
  - new discard_zeroes_data, discard_granularity, discard_alignment flags
  - Discard now carries a single page payload that the driver can use for its purposes

- Linux 2.6.36 (2010) adds support for secure erase into the discard code, and leaves payload allocation to the driver
History of discard in Linux (3)

- Linux 2.6.37 (2011) removes the barrier semantics and makes discard synchronous.
- Linux 2.6.38 (2011) adds the FITRIM ioctl to discard all free space in a file systems.
- Each release more file systems start issuing online discards.
Online discard in XFS

How do file systems free blocks?

- Needs to be atomic vs deleting them from the extent list
  → Atomic transaction that logs the intent to free, actual freeing delayed

- Transactions might be asynchronous
  → Must only reuse or discard blocks once actually committed
The busy extent list

- Tracks all extents that have their deletion intent committed but the transaction not safely on disk yet
  - Red / Black tree per allocation group
  - Allocations try to skip busy extents when possible
  - If not the transaction freeing them has to be forcibly written to disk
The busy extent list - discards

- Reuses the busy extent list:
  - Once the transaction committing the deletion is on disk, issue a discard for all deleted extents
  - Extents stay on the busy extent list
  - Only get removed once the discard completes
  - Initially discards were issued synchronously → blocks the log write completion thread

- As part of discard support the busy extent list was improved:
  - Scalable and bulletproof *(at least we thought..)*
Asynchronous discards in the file systems

- Do not wait for the discards from the log write completion handler
  - Instead attach a completion handler that removes them from the busy extent list
  - Forces us to wait for discards in various places, including the near ENOSPC allocator code
  - Ended up finding lots of bugs in this code
Recent discard improvements

- Linux 4.7 adds usable asynchronous discards supports
  - Allows for attaching a completion callback

- Linux 4.10 improves the way they payloads for TRIM / UNMAP / WRITE SAME are allocated
  - Doesn’t pretend to be the normal I/O path
  - Special drivers overrides the payload path now
Ranged TRIM support

- Linux so far only allowed a single discard range
- Linux block I/O requests generally are LBA-contiguous, although multiple bios can be merged into one
  - Ranged discard uses this linkage to allow linking non-contiguous bios for discard if the driver allows it
  - Driver then walks the list of bios and generates the payload
  - Multiple ranges only happen when issued asynchronously
- Linux 4.11 supports ranged deallocated for NVMe
ATA ranged TRIM support

- Libata translated SCSI into ATA commands
- For discards it advertises **WRITE SAME** support and builds **TRIM** commands
  - **WRITE SAME** only supports a single range
  - TRIM supports multiple small ranges
  - In SCSI **UNMAP** would support multiple ranges, but the semantics don’t match very well
  - Rewriting the payload in place corrupts user data for SCSI pass through
ATA ranged TRIM support (2)

- Maybe we should get out the command rewriting business?
  - Add a new Vendor Specific SCSI command with the ATA TRIM payload
  - Greatly simplifies the libata code
  - Discard can now use the zero page as WRITE SAME payload
- Submitted for Linux 4.12, not merged yet
NVMe enterprise SSD (Vendor A)

![Bar chart showing performance metrics for NVMe SSDs]

- **4.10, no discard**
- **4.10 discard**
- **4.11 no discard**
- **4.11 discard**

Legend:
- `rm -rf + sync (in seconds)`
NVMe enterprise SSD (Vendor B

- 4.10, no discard
- 4.10 discard
- 4.11 no discard
- 4.11 discard

```
rm -rf + sync (in seconds)
```

<table>
<thead>
<tr>
<th>Version</th>
<th>Status</th>
<th>Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.10, no discard</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4.10 discard</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>4.11 no discard</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4.11 discard</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
SATA SSD (non-queued TRIM)

- 4.10, no discard
- 4.10 discard
- 4.11+ no discard
- 4.11+ discard

Title

rm -rf + sync (in seconds)
(Ab)using discard for zeroes

- WRITE SAME guarantees that future reads return all zeros.
  - Wouldn’t it be nice to use that for zeroing?
- Keyed off the discard_zeros_data flag
  - Works perfect for WRITE SAME
  - But now discard isn’t just a hint any more
  - Failure reporting becomes important now, e.g. for too small or unaligned requests
More Zeroing offload

- Linux 3.7 (2012) adds support for explicit WRITE SAME operations
  - can be used for zeroing without the UNMAP bit
- Linux 4.10 (2017) adds an explicit zeroing operation (**REQ_OP_WRITE_ZEROES**)
  - No payload (same as discard)
  - Can be implemented directly (NVMe)
  - Or by adding a payload (e.g. SCSI)
Questions?