Design Document
For the
Client In-Memory Checksum

Revision History

Date	Revision	Author
11/02/2011	Original	J. Xiong
11/14/2011	0.9	J. Xiong
11/29/2011	0.91 - minor updates	A. Dilger
11/29/2011	1.0	J. Xiong

Scope Statement

Client In-Memory Checksum Design Document

12/12/2011	1.1	J. Xiong

Introduction

Lustre has a feature to use checksum to verify data integrity in RPC transferring over network. However, the data checksum feature is not completely reliable in case of mmap writes because Lustre does not prevent mmap pages from being modified while they are being transferred, after the checksum has been calculated. This can cause a checksum mismatch on the OST when the page is verified. As a temporary solution, an OBD_FL_MMAP flag was introduced to the bulk write RPC so that clients can use it to notify the OST if the pages in the RPC come from an mmap file and might have an invalid checksum when they arrive. If a checksum mismatch is detected on an RPC with the OBD_FL_MMAP flag set, no error is printed on the console, and the OST requests that the client send the data again.

In newer Linux kernels(since 2.6.30), the vm_operations_struct::page_mkwrite() method is introduced to allow filesystems to be notified before a page is modified. This ensures that operations such as checksums, RAID parity calculations, etc to be done without risk of page modifications. We can use it to solve Lustre checksum problem over mmap also.

Because it consumes lots of CPU power to calculate the checksum, it will be better to distribute this workload to all cores in the system. This problem is partly addressed in ORNL-22, but we will cover it in this document for completeness.

Definitions

Checksum: In this document, **Checksum** means RPC checksum, a mechanism in Lustre to make sure the data is not mangled over network;
page_mkwrite(): abbreviation for vm_operations_struct::page_mkwrite;
page_fault(): abbreviation for vm_operations_struct::fault;
PTE: Page Table Entry;
WP: Write Protect, used by PTE to mark if a page is read only;
PG_dirty: Dirty bit of page;
PG_writeback: Write back bit of page.

Requirements

Make checksums work properly for mmap pages

By implementing page_mkwrite() method, Lustre can be notified before a mapped page is going to be modified.

Distribute checksum calculation workload to all CPUs

Calculating the bulk checksum consumes a lot of CPU time. To minimize the checksum overhead for single threaded writes, we should distribute the workload of calculating checksums to all CPU cores to fully utilize available compute cycles.

Functional specification

In order to satisfy the above requirements, the pages in RPC should be prevented from being modified until the RPC is finished. Also we need an implementation of multiple ptlrpcd to distribute checksum calculating to all CPU cores. I'm going to discuss them in detail in the following sections.

Make page content immutable during transfer

Making pages immutable before checksum calculation is necessary. Any change to the page content after checksum calculation means the OSTs can't distinguish if the page was modified on the client during transfer, or if it was corrupted by the network. We should block any further modification of pages after RPC checksum is calculated.

When Lustre is flushing a page to the OST, it will lock the page, call clear_page_dirty_for_io() to the page, and then set PG_writeback. The PG_writeback flag will be cleared when the RPC is finished. One thing is worth mentioning that in clear_page_dirty_for_io(), the kernel will not only clear the page dirty bit, also for every PTE in which the page is mapped it will clear PG_dirty and set WP. This will block further writes to pages during transfer and cause the ->page_mkwrite() method to be called on the next write.

In the current Linux implementation, a page can be written in two ways: from VFS, or VM if the page is mmapped into processes' address space.
For VFS write, since we usually hold page lock during transfer, the process will

be blocked until page lock is released. For PG_writeback support (added in ORI-279), it will wait for page writeback in ll_{prepare_write|write_begin}, which is all right too.

For writes from VM, page_mkwrite() will help us address the problem. Whenever a page is to be firstly dirtied via VM, this method will be called by kernel and if the page is being transferred we can wait for PG_writeback there, and then add this page into dirty cache.

Blocking VFS writes to a page during checksum calculations has already been implemented in current implementation of Lustre. Thus we will focus on blocking writes from VM in the rest of this document.

Distribute checksum workload to all CPUs

Calculating checksums consumes considerable CPU time on the client. To fully utilize the power of multi-core CPUs, we will take advantage of multiple ptlrpcd threads (ORNL-22) and calculate checksum in ptlrpcd thread context.

However, this is a double-edged sword. Though using multiple CPU cores helps calculate checksums faster, it may lower the overall system performance in some cases. For example, in non-uniform memory architectures (NUMA), it will consume a lot of system bus bandwidth for CPUs to access memory from remote NUMA nodes. Another disadvantage of calculating the checksum in ptlrpcd context is that it will pollute CPU data cache. To minimize cache pollution, ideally we should calculate the checksum immediately after the write is complete, that's in ll_{commit_write|write_end}.

Use Cases

Write a file via mmap

When a file is being written with checksum enabled, and if the OST sees a checksum error on the server side, it means the data is corrupted during transfer. The checksum must be correct with mmap or VFS writes, or both.

Logic specification

This section describes how we use page_mkwrite() to make page content

immutable during RPC, also describe a little bit about multiple ptlrpcd for completeness.

Block writes from VM if a page is in RPC

As described earlier, page_mkwrite() method is called if a page is being modified for the first time. This is because clean pages should have the WP set so that CPU will page fault in the kernel whenever applications try to modify those pages. This implies that page_mkwrite() may be called multiple times for a single page, since the page may have been mapped to multiple processes, or even multiple times in the same process.

Now that page_mkwrite() and page_fault() share almost the same kernel parameters, we will adopt the same IO type CIT_FAULT to handle page_mkwrite() in Lustre. A special flag, VM_FAULT_MKWRITE is set by kernel in vm_fault() for page_mkwrite() so that we can distinguish them.

Peek a cl_lock for page_mkwrite()

We need to handle a race where a page has already been truncated due to lock revoke, when the page is being called with page_mkwrite(). Under this situation, it makes no sense for page_mkwrite() to enqueue new RW cl_lock because kernel will finally find out the PTE for this page has ever changed, therefore it will reinstall the PTE with page_fault() and page_mkwrite() will be called again to make it writable. The method cl_io_lock_dmd::CILR_PEEK is invented to solve this race.

With CILR_PEEK, instead of calling cl_lock_request() in cl_io_lock(), cl_lock_peek() will be called. If the coresponding write lock covering this page is already/being revoked, cl_lock_peek() will return –ENODATA so that page_mkwrite() will detect this case and return VM_FAULT_NOPAGE to kernel so that kernel will redo the fault.

vvp_io_fault_start() for page_mkwrite()

In vvp_io_fault_start(), we will distinguish if it's called from page_mkwrite() by checking VM_FAULT_MKWRITE flag in vm_fault(). If it is the case, it will wait for PG_writeback to be cleared by calling wait_for_page_writeback(), and then

add the page into dirty cache by calling cl_page_cache_add() so that it can reserve space on the OST earlier.

A problem arises if cl_page_cache_add() returns an error due to insufficient grants or lack of quota. In this case, it means we can't cache this page so NOPAGE will be returned to kernel. We need to handle grants carefully to avoid a false ENOSPC error. However, this specific discussion is not in the scope of this document.

Implementation of multiple ptlrpcd threads (ORNL-22)

This functionality was implemented in preparation for the multi-core checksum functionality, and has already been merged into Lustre 2.2. For completeness, a brief summary will be given of how it works and how it helps distribute the checksums to all CPUs in the system.

In the new scheme of multiple ptlrpcd, there are as many as the number of CPU cores ptlrpcd threads started at system initialization time; and the caller can select one of the following policies so that the corresponding request can be handled by specific ptlrpcd threads:

```
typedef enum {
        /* on the same CPU core as the caller */
        PDL_POLICY_SAME         = 1,
        /* within the same CPU partition, but not the same core as caller */
        PDL_POLICY_LOCAL        = 2,
        /* round-robin on all CPU cores, but not the same core as caller */
        PDL_POLICY_ROUND        = 3,
        /* the specified CPU core is preferred, but not enforced */
        PDL_POLICY_PREFERRED    = 4,
} pdl_policy_t;
```

PDL_POLICY_ROUND will be used for BRW requests, so they will be processed by other ptlrpcd threads in a round robin pattern. Depending on benchmark results, it may be better to use PDL_POLICY_LOCAL if cross-CPU memory access causes a performance impact.

Another adjustable policy for multiple ptlrpcd is the bind policy which controls how to assign ptlrpcd to CPU cores. The following policies can be applied via module parameters "ptlrpcd_bind_policy":

```
typedef enum {
        /* all ptlrpcd threads are free mode */
        PDB_POLICY_NONE         = 1,
        /* all ptlrpcd threads are bound mode */
        PDB_POLICY_FULL         = 2,
        /* <free1 bound1> <free2 bound2> ... <freeN boundN> */
        PDB_POLICY_PAIR         = 3,
        /* <free1 bound1> <bound1 free2> ... <freeN boundN> <boundN
free1>,
        * each ptlrpcd[X] has two partners: thread[X-1] and thread[X+1]*/
        PDB_POLICY_NEIGHBOR     = 4,
} pdb_policy_t;
```

With bound mode, a ptlrpcd thread is bound to a specific CPU at initialization time; with free mode, the ptlrpcd thread can be scheduled to any free CPU. By default, PDB_POLICY_PAIR is used. There are two threads for a CPU core. When an RPC is assigned to a CPU, if the bound mode thread is busy, the RPC will be assigned to the free mode thread. Please contact Fan Yong <yong.fan@whamcloud.com> for more information.

Issuing RPC asynchronously in write path

We used to check and send RPCs in commit write path, this will cause checksum computation in the writing process. In order to handle thread write and checksum in parallel as much as possible, the process of writing a page is changed slightly. In osc_queue_async_io(), instead of calling osc_check_rpcs() directly, we're going to wake up ptlrpcd by queuing a fakereq, so that brw_interpret() will be called and send RPCs asynchronously, in ptlrpcd context.

Wire protocol changes

No new wire protocol changes are needed. Once the mmap pages are immutable during checksum operations, there will be no need to pass OBD_FL_MMAP for writes from mmap pages. This is already the case for normal page writes and does not need any change in OST behaviour. The OBD_FL_MMAP flag will continue to be supported on the OST for

compatibility with older clients.

Open issues and future work

The PDL_POLICY_LOCAL implementation does not currently have NUMA aware scheduling that would allow the caller to restrict checksum operations to the ptlrpcd threads on the same NUMA core. Until that is implemented as part of the SMP optimization project, PDL_POLICY_LOCAL behaves the same as PDL_POLICY_ROUND. After the SMP patches are landed, we have to get back to benchmark each policy and may find different policies for different job type.