LLOG CTXT REFERENCE COUNT HLD(BUG 10800)

Author WangDi

2007/3/14

1 Introduction

In current implementation, when llog is cleanup, the ctxt is just freed instead of checking whether other users using it. So there might be the race between llog cleanup and other llog user threads. Llog ctxt refcount will be introduced to protect ctxt from being freed improperly during llog cleanup .

2 Requirement

- Handle the race (llog cleanup vs other llog user thread) properly after llog ctxt refcount is introduced.
- Any user must increase the refcount of the llog ctxt before using the llog, and decrease the refcount after using it.
- Any log API interface should not be modified.

3 Definition

3.1 Synchronous LLog user

Synchronize llog user means those llog users who will use llog ctxt in req handler or setup/cleanup process(process config log) synchronously.

3.2 Asynchronous Llog user

Asynchronous llog user means those llog users who will use llog ctxt in a separate thread, other than request handler or setup/cleanup process.

4 Functional specification

4.1 Data structure

1. Llog ctxt refcount will be added to indicate how many users are using this llog.

```
\begin{lstlisting}
    struct llog_ctxt {
    .....
    atomic_t loc_refcount;
    };
    \end{lstlisting}
```

1. The llog cleanup waitq will be added to the obd_device in case the cleanup process waits for other llog users decrease the refcount of the ctxt and finally releasing(freeing) it.

```
\begin{lstlisting}
    struct obd_device {
    .....
    wait_queue_head_t obd_llog_waitq;
    };
    \end{lstlisting}
```

4.2 Function specification

1. llog_get_context(struct obd_device *obd, int index)

   Obd is the obd device the llog ctxt locating, index is the ctxt index. Whenever beginning to operate the llog indicated by the obd and index, the user calls this API to increase the refcount of the llog ctxt.

2. llog_put_context(struct llog_ctxt *ctxt)

   When finishing using the llog, the user calls llog_ctxt_put to decrease the refcount of the ctxt and release(free) it if the ref count of the llog reaches to 0.

5 Use cases

After llog_get/put_context is introduced, the llog user should call llog_get_context to increase the ctxt refcount before accessing it, llog_put_context will be called to decrease the ctxt refcount after using it. For example, for mds_log_op_unlink

```
\begin{lstlisting}
int mds_log_op_unlink()
{
.....
```

```
struct llog_ctxt *ctxt;
.....
ctxt = llog_get_context(obd, LLOG_MDS_OST_ORIG_CTXT);
rc = llog_add(ctxt, &lur->lur_hdr, lsm, logcookies,
cookies_size / sizeof(struct llog_cookie));
llog_put_context(ctxt);
.....
}
\end{lstlisting}
```

Currently, there are following llog users. **Note: In these llog users, only those asynchronous llog users will be indicated, which might be race with llog cleanup which will be discussed in State Management section.**

5.1 LLOG SERVE

- llog_origin_handle_create: it will call llog_get/put_context to get the llog ctxt and handle llog object create req.
- llog_origin_handle_destroy: it will call llog_get/put_context to get the llog ctxt and handle llog object destroy req.
- llog_origin_handle_next_block: it will call llog_get/put_context to get the llog ctxt and handle llog read next block req.
- llog_origin_handle_prev_block: it will call llog_get/put_context to get the llog ctxt and handle llog read previous block req.
- llog_origin_handle_read_header: it will call llog_get/put_context to get the llog ctxt and handle llog read header req.
- llog_origin_handle_cancel: it will call llog_get/put_context to get the llog ctxt and handle llog cancel req.
- llog_catinfo_deletions: it will call llog_get/put_context to handle showing unlink log req.

5.2 MDS

- mds_precleanup: it will call llog_get_context to get the correspondent ctxt and cleanup it. **Note: this get_context will be balanced by llog_put_context inside llog_cleanup, so llog_cleanup will have two put, one is for balance this context get, the other one is for put the refcount owned by this obd (the refcount to be initialized to be 1 to indicate the ctxt being used by this obd), which is similar for all the cleanup process.**
- mds_postrecov, mds_notify: it will call call llog_get/put_context to get the ctxt and make sure the ctxt is not NULL.

- __mds_lov_synchronize: it will call llog_get_context to get the llog ctxt to connect the OST for unlink log recovering. **Note: it is an asynchronous llog user.**
- mds_io_control: it will call llog_get/put_context to get the ctxt to handle those llog ioctl in mds.
- mds_llog_origin_add, mds_llog_origin_connect, mds_llog_repl_cancel: they will call call llog_get/put_context to get the llog ctxt to handle these mds llog API.
- mds_log_op_unlink, mds_log_op_setattr: they will call llog_get/put_context to get the llog ctxt to add the unlink/setattr log.
- mds_llog_finish: it will call llog_get/put_context to get the llog ctxt and cleanup them, similar as mds_precleanup.
- mds_join_file: it will call llog_get/put_context to get the llog ctxt and handle join request.

5.3 MGS

- mgs_precleanup: mgs will call llog_get_context to get the correspondent ctxt and cleanup it and similar as mds_precleanup.
- mgs_iocontrol: mgs will call llog_get_context to get the correspondent ctxt and handle these llog ioctl.
- mgs_get_fsdb_from_llog, mgs_modify, record_start_log, mgs_log_is_empty and mgs_erase_log: mgs will call llog_get/put_context to get the llog ctxt to handle various llog operation.

5.4 OSC

- osc_setinfo_mds_conn_interpret: it will call llog_get/put_context to get the correspondent(unlink) llog ctxt and init the import for sending recovery unlink log to OST.
- osc_llog_finish: it will call llog_get_context to get the correspondent llog and cleanup it.
- osc_disconnect: it will call llog_get/put_context to get the llog ctxt and sync the unflushed unlink log to OST.
- osc_mds_ost_orig_logops: MDS use these llog ops to add/cancel unlink log for each OST. MDS will call these log operations from LOV layer, and LOV layer will call llog_get/put_context to protect the ctxt in this layer.

5.5 OBDFILTER

- filter_cancel_cookies_cb: it will call llog_get/put_context to get the llog context to send cancel the llog cookie to MDS and running in an separate thread in OST. **Note: It is an asynchronize llog user**.
- llcd_send: it is a separate thread and to send the cancel llog cookie to MDS. It will use llog ctxt(loc_imp) when sending the cookie. **Note: It is an asynchronism llog user.**
- llog_recovery_generic: it will call llog_get/put_context to get the llog context and retrieve the cancel log from MDS and unlink orphans. It is running an separate thread in OST. **Note: It is an asynchronism llog user.**
- filter_llog_init: it will call llog_get/put_context to get the llog ctxt and set recovery unlink callback of the unlink llog ctxt.
- filter_llog_finish: it will call llog_get_context to get the llog ctxt and cleanup it.
- filter_disconnect, filter_sync: it will call llog_get/put_context to get the llog ctxt and flush any remaining cancel unlink log to MDS.
- filter_destroy: it will call llog_get/put_context to get the llog ctxt and send the cancel unlink log if the object is already gone.
- filter_set_info_async: it will call llog_get/put_context to init the import of unlink ctxt import.

5.6 LOV

- lov_llog_origin_add, lov_llog_origin_connect, lov_llog_repl_cancel : These llog API are used to help mds distribute its llog operations to each osc. They will call llog_get/put_context to get the llog ctxt of each OSC and hold the refcount of it, then do the operations, then put the llog ctxt.
- lov_llog_finish: it will call llog_get_context to get the llog ctxt and cleanup it.

5.7 MGC

- mgc_llog_init: it will call llog_get/put_context to get the llog ctxt and init the ctxt import.
- mgc_llog_finish: it will call llog_get_context to get the llog ctxt and cleanup it.

- mgc_process_log: it will call llog_get_context to get the remote config ctxt and try to retrieve the config log from mgs server and copy it to the local log if possible then call llog_get_context to get the local log ctxt, then process these config log, finally, call llog_put_context to put these llog ctxt.

5.8 LLOG_TEST

- llog_test_N: these llog test API will call llog_get/put_context to get the llog ctxt and do various of llog test.

6 Logic specification

6.1 LLog_get_context

In the llog_get/put_context, the ctxt refcount will be increased. Before increasing the refcount of ctxt, we should check whether the correspondent obd_llog_ctxt entry is NULL, if it is NULL, it means the llog ctxt is being freed at that time, and it should return NULL. If it is not, then the ctxt refcount could be increase, and return the ctxt. **Note: to prevent llog cleanup process just intruding between checking obd_llog_ctxt and llog_ctxt_get, these two steps should be protected with spin_lock(obd_dev_lock).**

```
\begin{lstlisting}
#define llog_ctxt_get(ctxt)
({
struct llog_ctxt *ctxt_ = ctxt;
LASSERT(atomic_read(&ctxt_->loc_refcount) > 0);
atomic_inc(&ctxt_->loc_refcount);
CDEBUG(D_INFO, "GETting ctxt %p : new refcount %d\n", ctxt_,
atomic_read(&ctxt_->loc_refcount));
ctxt_;
})
static inline struct llog_ctxt *llog_get_context(struct obd_device *obd, int
index)
{
.....
spin_lock(&obd->obd_dev_lock);
if (obd->obd_llog_ctxt[index] == NULL) {
spin_unlock(&obd->obd_dev_lock);
CWARN("obd %p and ctxt index %d is NULL \n", obd, index);
return NULL;
}
ctxt = llog_ctxt_get(obd->obd_llog_ctxt[index]);
spin_unlock(&obd->obd_dev_lock);
return ctxt;
```

```
}
\end{lstlisting}
```

6.2 llog_put_context

In llog_put_context, it will decrease the ctxt refcount, if the refcount is zero, the ctxt will be freed and wake up the process waiting for this ctxt. **Note: to prevent other llog users access the llog ctxt between the refcount become zero and freeing the ctxt, we will obd_llog_ctxt entry to NULL after refcount become 0, and put these two steps under spin_lock(obd_dev_lock).**

```
\begin{lstlisting}
static void llog_ctxt_destroy(struct llog_ctxt *ctxt)
{
.....
idx = ctxt->loc_idx;
obd = ctxt->loc_obd;
if (ctxt->loc_exp)
class_export_put(ctxt->loc_exp);
OBD_FREE(ctxt, sizeof(*ctxt));
.....
}
int __llog_ctxt_put(struct llog_ctxt *ctxt)
{
.....
obd = ctxt->loc_obd;
spin_lock(&obd->obd_dev_lock);
if (!atomic_dec_and_test(&ctxt->loc_refcount)) {
spin_unlock(&obd->obd_dev_lock);
return 0;
}
obd->obd_llog_ctxt[ctxt->loc_idx] = NULL;
spin_unlock(&obd->obd_dev_lock);
if (CTXTP(ctxt, cleanup))
rc = CTXTP(ctxt, cleanup)(ctxt);
llog_ctxt_destroy(ctxt);
wake_up(&obd->obd_llog_waitq);
return rc;
}
#define llog_put_context(ctxt)
do {
.....
__llog_ctxt_put(ctxt);
} while (0)
\end{lstlisting}
```

6.3 llog cleanup

In llog_cleanup, if the ctxt refcount is not zero, which means other llog users are using this ctxt, the cleanup process will be added to a waitq(obd_llog_waitq) and wait other users release the ctxt. **Note: we can not make cleanup process go on to destroy the obd device without waiting other users releasing the ctxt, because only keep the ctxt for other llog user thread is not enough, and they may also need a health obd. So cleanup process must wait all the llog user release the ctxt and free it, then continue.**

```
\begin{lstlisting}
int llog_cleanup(struct llog_ctxt *ctxt)
{
.....
/*Note: this put is for banlancing the ctxt_get when calling llog_cleanup
*/
llog_put_context(ctxt);
/* check whether the obd was cleanup */
spin_lock(&obd->obd_dev_lock);
LASSERT(obd->obd_stopping == 1);
spin_unlock(&obd->obd_dev_lock);
idx = ctxt->loc_idx;
/*release the ctxt of its own obd and
*try to free the ctxt inside __llog_ctxt_put*/
rc = __llog_ctxt_put(ctxt);
l_wait_event(obd->obd_llog_waitq, obd->obd_llog_ctxt[idx] == NULL,
&lwi);
RETURN(rc);
}
\end{lstlisting}
```

Note: Once those llog users detect the llog is cleanup(obd->obd_stopping == 1) , it should stop immediately and release the llog ctxt.

7 State Management

Currently, there are two kinds of llog users,

7.1 Synchronize llog user

There are two kinds of Synchronize llog users:

- For those users, who access the llog ctxt in req handler synchronausly. Because when handling req, the refcount of obd export (also the exp_rpc_count) will be increased to protect this export being disconnected. In the other hand, in obd cleanup process, all the exports should be disconnected before llog_cleanup, which will make sure all these synchronize llog user will

finished before llog_cleanup. So there will be no ctxt free race for this kind of case.

- Another kind of synchronize llog user are setup/cleanup process. For clients, we will use "mount" to mount client and vfs mount mechanism will make sure mount and umount will not happened in the same time, which means config llog processing will not happened in the same time with llog cleanup, so we do not need consider llog cleanup race for client setup. As for server setup, there are two kinds of situation

  - For new mountconfig, MDS/OST will call mgc_process_log to process the config log, so we should check whether the ctxt is NULL after get it by llog_get_context, in case it is being cleanup.
  - For old zero config, only mds will use config log and it will call class_config_parse_llog to process the config log in mds_postsetup, so we should also check whether the ctxt is NULL there.

7.2 Asynchronize llog user

Asynchronize llog users mean those users who use llog ctxt in a separate thread asynchronously. Because we do not have synchronize mechanism between these users and llog cleanup, the race might happen between them. So when these asynchronize users access the llog ctxt, they should check whether the llog ctxt is releasing. If it's not releasing, the user should call llog_get_context to increase the refcount and prevent the ctxt being released when using it, and call llog_put_context to put the refcount after using it. Currently, there are four kinds of asynchronize llog users:

1. Filter llog llog cleanup vs llog recovery process

   (a) When Filter setup, it tries to get recovery log (unlink log) from MDS and processes these llogs in a separate thread. llog_context_get should be called to increase the ctxt refcount. Note: If the obd is stopping, it should stop accessing the llog ctxt and return immediately.

   (b) If the filter is cleaning up(filter llog cleanup) before the llog recovery thread stops, the cleanup process will wait log recovery thread stop and release the ctxt, then continue.

   \begin{lstlisting}

   static int llog_recovery_generic(struct llog_ctxt *ctxt, void *handle,void *arg)

   {

   .....

   if (obd->obd_stopping) {

```
up(&llpa.llpa_sem);
RETURN(-ENODEV);
}
llpa.llpa_ctxt = llog_get_context(ctxt->loc_obd, ctxt->loc_idx);
if (!llap.llpa_ctxt) {
up(&llpa.llpa_sem);
RETURN(-ENODEV);
}
.....
}
\end{lstlisting}
```

2. Filter llog cleanup vs llog cancel cookie callbacks

   (a) In filter objects destroy commit callback, the filter will send the unlink log cookie back to MDS. Before accessing the llog, llog_ctxt_get should be called to increase the ctxt refcount.

   (b) If the filter was clean up before the callback thread stop, the cleanup process will wait the callback thread stop release the ctxt, then continue.

```
\begin{lstlisting}
    void filter_cancel_cookies_cb(struct obd_device *obd, __u64 transno, void
*cb_data, int error)
    {
    .....
    if (obd->obd_stopping) {
    OBD_FREE(cookie, sizeof(*cookie));
    return;
    }
    ctxt = llog_get_context(obd, cookie->lgc_subsys + 1);
    if (!ctxt) {
    OBD_FREE(cookie, sizeof(*cookie));
    return;
    }
    .....
    }
    \end{lstlisting}
```

1. Filter llog cleanup vs log_commit_thread

   Log commit thread is running in the OSS level, and it maintains a list of llcd, and each llcd item will access the attached llog ctxt without checking whether it is freed, so there is a race between this thread with llog_cleanup.

   (a) llog_get_context should be called to protect itself before the ctxt is attached to the llcd items. **Note: log_commit_thread can not make sure each llcd item will be put until the OSS service stop(ptlrpc module exit in oss level), but it happened after llog_cleanup. So we should move log_commit_thread to obdfilter level, and each obdfiler will have their own log_commit_thread. And before the llog_cleanup, it should stop this thread to avoid the race. Then we may need put llog_commit_master to the obdfilter and call llog_cleanup_commit_master in filter_cleanup. There is another method discussed in Alternative methods.**

   (b) Check whether the ctxt is being freed(checking obd_stopping flags, obd_stopping is set before llog_cleanup), when adding the llcd to the send list.

   (c) When the llog_commit_thread is stopped, all the llcd and its ctxt will be put.

   \begin{lstlisting}

   llog_obd_repl_cancel()

   {

   .....

   llcd = llcd_grab();

   llcd->llcd_ctxt = llog_get_context(ctxt);

   .....

   }

   llcd_put()

   {

   llog_put_context(llcd->llcd_ctxt);

   }

   /* Do not add llcd items to the sending llcd list, if the obd is stopping */

   void llcd_send(struct llog_canceld_ctxt *llcd)

   {

   /* this llcd_ctxt is protect by llog_get_context, so no need checking NULL pointer */

if (llcd->llcd_ctxt->loc_obd->obd_stopping)

return;

.....

}

\end{lstlisting}

2. MDS cleanup vs MDS llog_connect

   (a) When MDS do llog_connect to OST in an separate thread(__mds_lov_synchronize), it should call llog_ctxt_get to increase the llog ctxt refcount.

   (b) If MDS was clean up before the thread(__mds_lov_synchronize) stop, the cleanup process will wait synchronize thread stop and release the ctxt, then continue.

\begin{lstlisting}

static int __mds_lov_synchronize(void *data)

{

.....

if (obd->obd_stopping)

RETURN(-ENODEV);

ctxt = llog_get_context(obd, LLOG_MDS_OST_ORIG_CTXT);

if (!ctxt)

RETURN(-ENODEV);

.....

}

\end{lstlisting}

8 Protocol, APIs, disk format.

LLog_ctxt_get/put should be called before/after accessing the llog. No wire protocols and disk format changes for this HLD.

9 Test Plan

Replay dual 17 will be used to test this case. Several replay_single test case will be also needed for testing those asynchronous llog user.

1. llog cleanup vs ost llog_recovery_thread. OBD_FAIL_TIMEOUT will be used for simulate the race between them.

\begin{lstlisting}
#define OBD_FAIL_OST_LLOG_RECOVERY_TIMEOUT 0x21f
static int filter_recov_log_mds_ost_cb(struct llog_handle *llh,
struct llog_rec_hdr *rec, void *data)
{
.....
if (OBD_FAIL_CHECK(OBD_FAIL_OST_LLOG_RECOVERY_TIMEOUT))
OBD_FAIL_TIMEOUT(OBD_FAIL_OST_LLOG_RECOVERY_TIMEOUT,
30);
.....
}
#test race llog recovery thread vs llog cleanup
test_59()
{
mkdir $DIR/$tdir
createmany -o $DIR/$tdir/$tfile-%d 800
replay_barrier ost
OBD_FAIL_OST_LLOG_RECOVERY_TIMEOUT 0x21f
unlinkmany $DIR/$tdir/$tfile-%d 800
do_facet ost "sysctl -w lustre.fail_loc=0x8000021f"
facet_failover ost
sleep 10
fail ost
sleep 30
do_facet ost "sysctl -w lustre.fail_loc=0x0"
$CHECKSTAT -t file $DIR/$tdir/$tfile-* && return 1
rmdir $DIR/$tdir
}
run_test 59 "test race llog recovery vs llog cleanup"
\end{lstlisting}

1. llog cleanup vs llog cancel cookie callback.

\begin{lstlisting}
#define OBD_FAIL_OST_CANCEL_COOKIE_TIMEOUT 0x221
void filter_cancel_cookies_cb()
{
.....
if (OBD_FAIL_CHECK(OBD_FAIL_OST_CANCEL_COOKIE_TIMEOUT))
OBD_FAIL_TIMEOUT(OBD_FAIL_OST_CANCEL_COOKIE_TIMEOUT,
30);
.....
}
#test race cancel cookie cb vs llog cleanup
test_59c()
{

```
# OBD_FAIL_OST_CANCEL_COOKIE_TIMEOUT 0x221
touch $DIR/$tfile
do_facet ost "sysctl -w lustre.fail_loc=0x80000221"
rm $DIR/$tfile
sleep 10
fail ost1
}
run_test 59c "test race mds llog sync vs llog cleanup"
\end{lstlisting}
```

1. llog cleanup vs llog commit_thread

   Because we will stop log_commit_thread, before llog_cleanup. So there is no race between log_cleanup and log_commit_thread.

2. llog cleanup vs MDS cleanup

```
\begin{lstlisting}
#define OBD_FAIL_OST_LLOG_RECOVERY_TIMEOUT 0x220
static int __mds_lov_synchronize()
{
.....
if (OBD_FAIL_CHECK(OBD_FAIL_MDS_LLOG_SYNC_TIMEOUT))
{
OBD_FAIL_TIMEOUT(OBD_FAIL_MDS_LLOG_SYNC_TIMEOUT, 60);
}
.....
}
#test race mds llog sync vs llog cleanup +test_59b()
{
# OBD_FAIL_MDS_LLOG_SYNC_TIMEOUT 0x137
do_facet mds "sysctl -w lustre.fail_loc=0x80000137"
facet_failover mds
sleep 10
fail mds
do_facet client dd if=/dev/zero of=$DIR/$tfile bs=4k count=1 || return 1
rmdir $DIR/$dir
}
run_test 59b "test race mds llog sync vs llog cleanup"
\end{lstlisting}
```

10 Alternatives.

We can also add some flags to indicate whether other threads are using the ctxt. But there are many kinds of llog user threads(as section 6 indicate), if we implement in this way, we might need separate flags for each user which will make things much complicated and out of control. So we choose using llog ctxt refcount to make things clear and easy.

Another method to resolve the race between llog_cleanup and llog_commit_thread is that

- Still put log_commit_thread in OSS level.
- Try to walk through the llcd_list and put the correspondent llcd item. But in current implementation, the llcd items was in three list in log_commit_thread:lcd_llcd_list, llcd_pending_list, llcd_resend_list, and unfortunately, lcd_llcd_list can not be accessed from outside of log_commit_thread. So we need create a new list for each import and track the llcd for the import. But it will bring much complication and trouble.