HLD of Parallel Lock Callback

Niu YaWei

2006-11-28

1 Introduction

Provide a mechanism to perform the LDLM lock callbacks in parallel.

2 Architecture

https://mail.clusterfs.com/wikis/cfs/intra/LaunchWorkSheets/ParallelCb

3 Requirements

- Provide a mechanism to perform the LDLM lock callbacks in parallel.
- Instead of threads use ptlrpcd to achieve this.
- Keep same semantics for timeouts, reply code as today.

4 External Functional specifications

4.1 Prototypes

Currently, LDLM server gathers the blocking/completion locks in a list then sends the blcoking/completion callbacks one by one, to make these callbacks sending efficient, we are going to use another thread - ptlrpcd to send them in parallel, and the ptlrpc request set APIs will be used for this purpose.

```
int ldlm_run_bl_ast_work(struct list_head *rpc_list)
```

This function will be changed to use ptlrpc request set APIs to send blocking callbacks in parallel.

```
int ldlm_run_cp_ast_work(struct list_head *rpc_list)
```

This function will be changed to use ptlrpc request set APIs to send completion callbacks in parallel.

```
int ldlm_server_blocking_ast(struct ldlm_lock *lock, struct ldlm_lock_desc *desc, void
```

The server blocking ast will be changed to use ptlrpc_set_add_req() to add request into request set.

```
int ldlm_server_completion_ast(struct ldlm_lock *lock, int flags, void *data)
```

The server completion ast will be changed to use ptlrpc_set_add_req() to add request into request set.

```
struct ptlrpc_request_set *ptlrpc_prep_set(void)
```

Prepare a request set to contain all the callback reqeusts. No need to change this API.

```
void ptlrpc_set_destroy(struct ptlrpc_request_set *set)
```

Destroy the request set. No need to change this API.

```
void ptlrpc_set_add_req(struct ptlrpc_request_set *set, struct ptlrpc_request *req)
```

Add one callback RPC into the request set. No need to change this API.

```
int ptlrpc_set_wait(struct ptlrpc_request_set *set)
```

Wait for all the replies in request set done. No need to change this API.

```
static int ldlm_cb_interpret(struct ptlrpc_request *request, void *data, int rc)
```

This new added request interpret callback is called when recieves reply of a ldlm blocking/completion callback, it'll handle the error code of reply and propagate the -ERESTART to the callback processing thread if necessary.

4.2 Layering of API's

All the APIs are in ptlrpc module.

4.3 How memory for variables / parameters is allocated

- The requests and request sets are allocated dynamically at run time.
- To avoid huge request set eat up memory, we can split it into several request sets with reasonable size and send them serializely.

4.4 Context in which the functions run

All the functions run in server thread context.

5 High Level Logic

The logic of ldlm_server_blocking_ast() looks like:

```
ldlm_server_blocking_ast(struct ldlm_lock *lock, struct ldlm_lock_desc *desc, void *dat
{
    ptlrpc_request_set *set = data;
    req = ptlrpc_prep_request();
    ptlrpc_set_add_req(set, req);
}
```

The logic of ldlm_run_bl_ast_work() looks like:

```
ldlm_run_bl_ast_work(list_head *rpc_list)
{
    set = ptlrpc_prep_set();
    list_for_each_safe(rpc_list) {
        lock->l_blocking_ast(lock, desc, set, LDLM_CB_BLOCKING);
    }
    ptlrpc_set_wait(set);
    ptlrpc_set_destroy(set);
    if (any of ldlm_cb_interpret returns -ERESTART)
        return -ERESTART;
}
```

The logic of ldlm_cb_interpret() looks like:

```
ldlm_cb_interpret(struct ptlrpc_request *req, void *data, int rc)
{
    /* handle error */
    rc = ldlm_handle_ast_error(lock, req, rc, ‘‘blocking’’);

    /* propagate the -ERESTART error by data */
}
```

The logic of processing completion callbacks is similar with the blocking callbacks'.

6 Use-Case Scenarios

- Apply this mechanism for sending blocking and completion callbacks.
- Needn't apply this mechanism for sending glimpse callback, since server just sending one glimpse callback to the right client at a time.

7 State Machine Design

In current LDLM, the lock enqueue thread may stall on sending blocking/completion callbacks when there is a huge blocking/completion list, since it always sends the callback RPCs one by one. With the new parallel callback mechanism, the lock enqueue thread will just add each request into a request set then go to sleep, ptlrpcd will manage to send the RPCs in the request set simultaneously, after all requests done, ptlrpcd wake up the lock enqueue thread to move on.

7.1 Locking

Using resource lock (lr_lock) carefully to protect the lists/flags of lock and resource.

7.2 Cache Usage

None.

7.3 Recovery

No effect.

7.4 Disk state changes

None.

8 Test Plan

We need the new obdecho which can simulate thousand of clients, liblustre process isn't suit for such simulation, since liblustre lock will be canceled on server instantly when blocking happens. The test plan is:

- echo client cache huge amount of PR locks firstly, then another client perform a conflicting PW lock enqueue, stats this PW lock's enqueue time. Perform this test with and without parallel callback mechanism, then compare the result.
- echo client enqueue a PW lock firstly, then another echo client simulate huge amount of conflicting PR lock enqueue requests, then first client cancel the PW lock, stats the total and average PR locks' completion time. Perform this test with and without parallel callback, then compare the result.

9 Plan Review