Platform independent proc interface

Author: WangDi & Komal

05/07/2008

1 Introduction

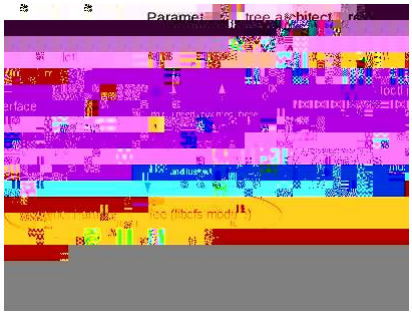
This document describes how to implement a platform independent proc interface for Lustre. The basic idea is that the platform-independent proc tree will be maintained in kernel base which is similar as linux proc tree(called parameters tree in this HLD). The tree will only be accessed by lctl params command with ioctl.

2 Requirements

- The parameters tree must be platform-independent, and similar as procfs in linux. Each entry is associated with a name, a value and correspondent functions for read/write. And any related platform-independent stuff should only in libcfs module.
- The parameters tree is used by both lustre and lnet, so it should be implemented in libcfs.
- The user tools could extract the data from the parameters tree with binary format, instead of a blob of text currently.

3 Functional specification

3.1 General Description



In the new implementation, lctl provides several APIs for acessing the parameter tree, described in 3.2. The name list in the request (lctl get/set_params) will be expanded by glob in these API, then sent to kernel parameters tree handler inside libcfs module by ioctl interface (might use socket when this parameters tree is in user space). In the handler, the params request will be handled by the callback of lustre/lnet registered in the tree.

3.2 API for lctl

These APIs are used by lctl to get/set value to the parameters tree.

3.2.1 Read and Write

```
int params_read(char *path, int path_len, struct list_head *value_list,
                int offset, int opt_flag);
int params_write(char *path, int path_len, struct list_head *value_list,
                int offset, int opt_flag);
void params_value_free(struct list_head *value_list)
```

- parameters
  - path: the path of the entry in the tree.
  - path_len: the length for the path.
  - value_list: the entry value list for set/get.
  - offset: offset for read/write.
  - opt_flag: indicates which option is set.
- Return
  - Read, >= 0 the read length, < 0 error.
  - Write, >= 0 the written length, < 0 error.
- Description
  - Read/Write APIs will be used by lctl get/set_params to set/get value of the parameters tree. params_value_free is used to free the list of params_entry.

3.2.2 List

```
int params_list(char *path_pattern, struct list_entry **list_entry);
void params_list_free(struct list_entry *list_entry);
struct list_entry
{
     char *le_name;   /* Note: here the le_name is the whole path name for the entry */
     int   le_name_len;
     struct list_entry *le_next;
     int   le_mode; /*indicate whether it is entry dir or entry*/
}
```

- parameters
  - path_pattern: The path_pattern of the list. It may includes some wild card characters, for example obdfilter.*OSC.stats.
  - list_entries: the list of matched entries in params_list. The list being freed in params_list_free.
- Return
  - = 0 success, < 0 error.
- Description
  - These API is used to get or free the lists of the entries.

3.3 API for parameters tree

Lctl uses ioctl to access the parameters tree. In the kernel base, the ioctl handler will be in libcfs module. Then both lnet and lustre need to register its own handler in libcfs_ioctl to handle the parameters tree ioctl command.

```
int libcfs_iocontrol(unsigned int cmd, void *arg);
```

- parameters
  - cmd: the ioctl command.
  - arg: buffer containing the input.
- Description
  - The API is used to handle ioctl request.
- Return
  - = 0 success, < 0 error.

3.4 Parameters tree in kernel base

As discussed, the parameters tree will be maintained in kernel base, which functionality is similar as procfs in linux kernel but we intend to make it platform independent unlike procfs which is linux kernel specific. The entries can be added/deleted/lookup in the similar way as its done with procfs interface.

3.4.1 params tree structure

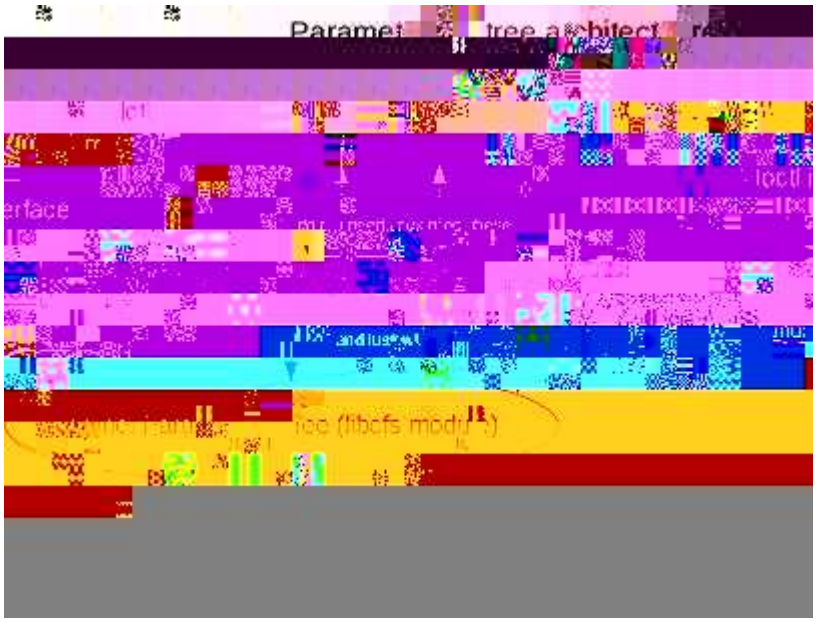
There will be a unique lustre_params_root (structure lustre_params_entry) for each server node. Each entry is associated with a name, a value and a corresponding read/write callback just like procfs in the linux kernel. The structure is also similar as a proc entry.

```
structure lustre_params_entry {
     struct lustre_params_entry   *lpe_subdir;  /*point to its first children */
     struct lustre_params_entry   *lpe_next;    /*point to its sibling, the end of this
     struct lustre_params_entry   *lpe_parent;
     lustre_params_read_t         lpe_cb_read;
     lustre_params_write_t        lpe_cb_write;
     atomic_t                     lpe_refcount;
     char                        *lpe_name;
     int                          lpe_name_len;
     rw_sem                      lpe_rw_sem;
     __u32                        lpe_version;
     void                        *lpe_data;    /* The argument for the read and write c
     int                          lpe_mode;    /* dir, file or symbol_link, and also th
     __u32                        lpe_magic;   /* Make sure the structure is valid */
};
```

typedef int (lustre_params_read_t)(char *page, char **start, off_t off, int count, int *eof, void *data);

typedef int (lustre_params_write_t)(struct file *file, const char __user *buffer, unsigned long count, void *data);

The structure of the params tree is shown in the figure below:



3.4.2 API for the parameters tree

There are two groups of API associated with the tree.

Updating API:

```
int params_add_entry(struct lustre_params_entry *lpe, char *name,
                     lustre_params_read_t *read_cb,
                     lustre_params_write_t *write_cb, void * data);
int params_delete_entry(struct lustre_params_entry *lpe, char *name);
struct params_entry *params_lookup_entry(struct lustre_params_entry *lpe, char *name);
```

- parameters
  - lpe: the parent for add/delete/lookup.
  - name: the name of the added/deleted/lookup entry. In delete_entry, if the name is NULL, it means it will delete the whole subtree under the lpe.
  - read_cb: the read callback for accessing the value attached to the entry.
  - write_cb: the write callback for accessing the value attached to the entry.
  - data: the parameters put to the lpe_data.
- Return

  - Read, >= 0 the read length, < 0 error.
  - Write. >= 0 the written length, < 0 error.
  - lookup, if it can find the entry according to the name, if it can not find, return NULL.

- Description
  - These 3 APIs will be used to add/delete/lookup the entry to the kernel based params tree.

4 Use cases

1. Set/get/list params tree parameters
   (a) Lctl set/get_params calls parameters lctl API to get/set parameters.
   (b) In kernel side, the ioctl request will be directed to libcfs module. And libcfs will call the lustre or lnet handler (registered in libcfs module) to handle the request.
2. Add/remove params tree entry
   (a) OBD calls lprocfs API to add/delete the entry of the tree.
   (b) In lprocfs code, params updating API will be called to add/delete entry of the tree.
3. Another important use case is the race between obd cleanup (params remove) and params accessing (lookup/read/write), which will be discussed in section 6 State management.

5 Logic specification

5.1 lctl interface

5.1.1 Interface structure

Because it requires to output the data with binary format, instead of a blob of string. So the following structure will be used to communicate between parameters tree and lctl command.

```
struct params_value_entry {
      enum params_value_type pve_type;
      __u32 pve_name_len;
```

```
        char *pve_name;
        __u32 pve_value_len;
        char *pve_value;
        char* pve_value[0]; /*could be buffer pointer or just interger depends on pv_type
};
```

When reading or listing entries, params kernel tree will packing the multi entries to the output buffer, then lctl will unpack the entry from the buffer.

5.1.2 lctl utility

Current lctl implements set/get_params interface based on several posix system calls like open, read, write, glob and close. All of them are based on local linux procfs. Since we need to achieve platform independence, these linux procfs dependency APIs need to be replaced. The lctl will implement get/set parameters by the APIs defined in Section 3.1. Inside these APIs, they will use ioctl interface to direct the correspondent parameters request to libcfs API.

- From the input provided by user, the path name and the options set will be seperated.
- Call params_list to get the matched entry lists according the path name.
- Call params_read/write to set/get the values of each entries of the list.

```
int jt_lcfg_getparam(int argc, char **argv)
{
    /* Analyze and retrieve the parameters from argc and argv */
     /* Retrieve all the list matched the list_path pattern */
      rc = params_list(list_path, &le);
      le_list = le;
      while (le) {
           param_read(le->le_name, le->le_name_len, op_buf, op_buf_len,
                      opt_flag);
           /*show result*/
           le = le->le_next;
      }
      /* free the params list */
      params_list_free(le);
      return 0;
}
```

5.1.3 read/write/list params for lctl

Since the path parameters in read/write_params is the exact path of the entry without wildcard characters, so we just need simply pack the correspondent parameters, and then call ioctl. Note: Here, obd_ioctl_data will still be used to here to pack the ioctl request, but the defination should be moved to libcfs.

```
int params_read(char *path, int path_len, char *read_buf, int buf_len, int offset)
{
    struct obd_ioctl_data data = { 0 };
   /* pack the parameters to data first */
    rc = obd_ioctl_pack(&data, &buf, sizeof(raw));
    /* open libcfs_dev_id and prepare for the following ioctl,
     * libcfs_dev_id should be registerd when lctl is initialized */
    rc = do_ioctl(libcfs_dev_id, LCTL_GET_PARAM, data);
    if (rc == 0)
        /* Success, data.out contains the output */
        /* unpack the params_value_entry from the buffer */
        /* Copy the output into read_buf */
    else
        //Failure, print the error.
    return rc;
}
```

As for params_list, because the path may include some wildcard characters and implementing wildcard characters match in kernel base would be unefficient and complicated, all the match logic would be implemented in params_list(user base) with the help of glibc reg match lib.

```
int params_list(char *path_pattern, struct list_entry **le)
{
      char *parent_path, look_name;
      int parent_path_len;
      struct fifo_entry *fifo; /*defined below */

      /*locate the wildcard characters in the path_pattern */

      /*Note: We can use FIFO list to implement the path wildcard match.
       *The entry in the fifo list:
       *Struct fifo_entry {
       *     char *parent;
       *     int path_len;
       *     char *left_wildcard;
```

```
         *     int left_wildcard_len;
         * }
         */
        /*locate first wildcard character, and add it to the fifo list */
        locate_wildcard(path_pattern, p_wc);
        add_to_fifo_entry(path_pattern, p_wc);
        do {
             /*1. Get the entry from fifo_list*/
             /*2. Read the sub-dir entries according to the parent of the entry, Note: he
             /*3. Check whether these sub-entries is matched with left_wildcard in the en

              for (le=sub_dir_list; le; le=sub_dir->le_next) {
                 if (matched left_wildcard) {
                    if (!entry->left_wildcard)
                         add_to_return_le(le, entry);
                    else
                         add_to_fifo_list(entry);
                 }
              }
        } while (!empty_fifo_enty(fifo_list));
}
```

5.2 Parameters tree in kernel base

5.2.1 General architecture

In current implementation, lustre modules use lprocfs interface(in obd_class) to access their procfs entry, where lprocfs is implemented based on linux procfs tree structure(proc_dir_entry) and linux procfs API. But because params_tree and linux procfs has similar structure, so lustre will still use this lprocfs interface to access the params_tree, to avoid to much code changes in lustre for new params_tree. For lprocfs, there should be only API name changes. But libcfs and lnet are below this layer(obdclass), so they will access the params_tree directly by the API described in following.

5.2.2 Parameters tree

1. Updating API

   These APIs are used to add/delete entries by other modules. The implementation should be simple, and it only need add/delete the entry links from the tree, but the process needs to be protected by the lpe_rw_sem in the parent.

```
struct lustre_params_entry * params_add_entry (struct lustre_params_entry *lpe,
                                               char *name,
```

```
                                                    lustre_params_read_t *read_cb,
                                                    lustre_params_write_t *write_cb,
                                                    void * data)
{
       /* create the child entry */
       obd_alloc_ptr(lpe_child);
       /* Fill lpe_child with write_cb/read_cb and data */
       /* Fill the structure of lpe */
       /* Here lpe_rw_lock will be used to protect the parent */
       down_write(lpe->lpe_rw_sem);
       /* link the lpe_child to the lpe children list
        * according to the figure of params tree structure in 3.3.1 */
       up_write(lpe->lpe_rw_sem);
       return 0;
}
int params_remove_entry (struct lustre_params_entry *lpe, char *name)
{
    /*Find child entry from lpe according to the name*/
    down_write(lpe);
    lpe_child = params_lookup_child_entry(lpe, name);
    /* unlink the entry from the tree */
    params_remove_child(lpe_child);
    up_write(lpe);
    return;
}
```

2. Accessing API

   These APIs are called to read/write/list the entries of the parameters tree. The general process of these API

   - Locate the entry according to the path.
   - Call read/write callback to get/set the value of the entry.
   - For list, it will pack the sub-entry name of this entry

   In these processes, lookuping the entry is similar as link_path_walk in linux kernel. Note: in the traversing process, when lookuping the children, the parent needs to be locked, and also the ref_count of the gotten child will be held, then the parent and child will be protected from being deleted in the process. The race will be discussed in section 6.

```
struct lustre_params_entry * params_lookup_entry (char *path)
{
       /*Got the name from each entry */
       struct lustre_params_entry *parent;
       struct lustre_params_entry *child = NULL;
```

```
        char *lookup_name;
        int lookup_name_length = 0, last_component = 0;
        parent = &lustre_params_root_entry;  /*initialize the root entry */
        name = path;
        /* Traverse the path and locate the entry, similar as link_path_walk,*/
        for (;;) {
             /*Get lookup_name lookup_name_length*/
             lookup_name = name;
             do {
                  c = *name++;
             } while (c && (c != '.')); /* path format looks xxx.yyy.zzz
             lookup_name_length = name - lookup_name;
             if (!c)
                    last_component = 1;
             /*lookup the name under parent */
             down_read(parent->le_rw_sem);
             /*Note: the found child should call lpe_ref_get(child)
              *to hold the refcount of the children*/
             child = lookup_entry(parent, lookup_name, lookup_name_length);
             up_read(parent->le_rw_sem);
             if (child == NULL)
                   break;
             if (last_component)
                   break;
             else {
                   lpe_ref_put(parent);
                   parent = child;
             }
        }
        return child;
}
```

For list, it need return all the children names. Because of limited output buffer size for ioctl, it also needs the offset and eof to indicate whether where to restart the listing and wheter listing is finished. So a dummy entry will be added to the params_tree to indicate the restart position of the list. The dummy entry will be skipped when others walking the tree.

```
int params_list_entry (char *path, __u64 offset, int *eof, void *buf, int buflen)
{
        /*Locate the entry by the path*/
        parent = params_lookup_entry(path);
        /*Locate the dummy entry(restart position) and restart list from that dummy
         *Note: the offset here is the lustre handle of the dummy entry.
         */
```

```
                /*pack the name to the buffer, until buffer is
                 *full or the end of the sub-entry*/
                /*Check whether it reaches the end of subdir,
                 *then set eof to tell the caller whether need another list */
        }
```

5.2.3 lprocfs interface

As discussed in 5.2.1, in lprocfs, the linux procfs API and dir_proc_entry need to be replaced with params tree API and lustre_params_entry.

- create_proc_entry : replaced with params_add_entry.
- remove_proc_entry: replaced with params_remove_entry.
- proc_dir_entry: replaced with lustre_params_entry

In lprocfs, seq_file is used to output the stats of the obd, where it will use seq_print to output the stats directly in kernel base. But with params_tree, the stats(lprocfs_counter array) will be returned to lctl, and output there.

5.2.4 seq file

Currently, some lustre proc entries use seq_file to output its “large” values to the user space, which can not be done in one time. In the new implementation, these seq_file entries will be changed to params_value_entry format, and output in “chunk” size to lctl one time. Since then we also need remember the offset to locate the restart position. Because seq_file should not be changed when it is being accessed, so the index could be used here to record the offset.

6 State management

Since the parameters tree might be accessed by several threads at the same time, lpe_rw_sem is brought into protect the tree.

- lpe_rw_sem
  - when lookup, get read_lock of the parent.
  - when add/delete entry, get write_lock of the parent.

Currently, lprocfs(lctl get/set_parameters) access lustre value by dp->data (proc_dir_entry), where data is usually obd_device. Then lprocfs use global lock(__lprocfs_lock) and dp_deleted flag, which indicate whether the entry is being deleted. With params tree, the global __lprocfs_lock will be replaced by the lpe_rw_sem locally in each entry. Here is the situation when raced is happening,

lctl get_params osc.lustre-OST0000-aaa.stats(pA) vs cleanup osc.lustre-OST0000-aaa(pB)

- pA: lprocfs got the entry osc by lookup in path traverse process.
- pA: It gets read_lock of osc, then lookup lustre-OST0000-aaa, hold its refcount.
- pB: Cleanup process locate the osc entry and waiting pA release the read_lock of osc.
- pA: Accessing the entry, release read_lock of osc.
- pB:get write_lock, unlink the entry and destroy it.

Note: In this process “lctl get_params–>lprocfs->params_tree->accessing obd_device”, the lock could only protect those values which are valid until obd_cleanup. If some varibles which are even changed(destoryed) before obd_cleanup, for example obd_import, special synchronisms are still needed here. But it is out of scope of this document.