# Userland proc interface

Author WangDI

05/07/2008

## 1 Introduction

This document describes how to implement proc and ioctl mechanisms on the user base server. The basic ideas are that the user base server maintain a platform-independent tree(similar as linux proc tree, it will be called parameters tree in this HLD) and lctl will retrieves those parames from the server, instead of procfs. And the communication between lctl process and user base server process will be implemented by socket, which will be implemented in another task.

Definition:

- lctl client: The process running lctl command. when it issues params requests to proc server.
- params server: The handler thread to handle the requests coming from the lctl client, which will be only running on the user base server in this HLD.

In this HLD, lctl client and params server are running are the same node.

## 2 Requirements

- The parameters tree must be platform-independent, and the entries in the tree could be added/deleted/lookup. Each entry in the the tree will associate with a name, value and read/write function. The value could be retrieved and set by the read/write function.
- The parameters tree is only maintained in user space server, and the correspondent lctl can be different from current procfs based lctl.
- The communication between lctl client and proc server is implemented by unix socket.

# 3 Functional specification

## 3.1 API for lctl

These APIs will be used by lctl to communicate with server process.

```
int params_open(char *dev_path);
int params_close(int sock_fd);
```

- parameters,
  - dev_path: connecting parameters(ports, connecting protocol) for the unix socket.
  - sock_fd: the descriptor for the socket.
- Return
  - open >0 the socket descriptor, < 0 error.
  - close = 0 success, < 0 error.
- Description
  - Lctl set/get_params will use params_open/close to get/close the socket descriptor for the communication.

```
int params_read(char *path, int path_len, char *read_buf, int buf_len,
int offset);
int params_write(char *path, int path_len, char *write_buf, int buf_len,
int offset);
```

- parameters
  - path: the path for params.
  - path_len: the length for the path
  - read_buf, write_buf: input/output buffer.
  - buf_len: the buffer length.
  - offset: the offset for read/write.
- Return
  - Read, >= 0 the read length, < 0 error.
  - Write. = 0 success, < 0 error.
- Description

- These 2 APIs will be used by lctl get/set_params to retrieve/set proc parameters.

```
int params_list(char *path_pattern, void *list_buf, int buf_len, int *real_len,
int *eof);
```

- parameters
  - path_pattern: The path_pattern of the list.
  - list_buf: the buffer to fill the listed entries.
  - buf_len: the length of the list_buffer.
  - real_len: the list_entry length in the list buffer.
  - eof: whether it is the end of listing.
- Return
  - $= 0$ success, $< 0$ error.
- Description
  - This API is used to get the lists of the entries.

```
int params_ioctl(int dev_id, int opc, void *ioc_buf, int ioc_buf_length);
```

- parameters
  - dev_id: the device id for ioctl request. For lustre, it should be a constant value as original (/dev/obd)
  - opc: the ioctl command.
  - ioc_buf: the ioc parameters buffer (obd_ioctl_data).
  - ioc_buf_length: the length for ioc_buf
- Return
  - $= 0$ success, $< 0$ error
- Description
  - This API will be used by lctl ioctl to do ioctl

### 3.2 API for params server

When the params server gets the request from the lctl client, it will setup a connection with lctl client, and then create another thread to handle the request. In the handler thread, it will call correspondent API according to the request. Except those API, all these stuff will be implemented in that communication task.

```
int params_server_read(int fd, char *path, int path_len, int offset);
int params_server_write(int fd, char *path, int path_len, int offset);
int params_server_list(int fd, char *path, int path_len);
```

- parameters
  - fd: the fd read/write will use to communicate with client.
  - path: the path for params, for list, the path might be a path_pattern string.
  - path_len: the length for the path.
  - offset: the offset for read/write.
- Return
  - $= 0$ success, $< 0$ error.
- Description
  - These 3 APIs will be used by params server to set/get/list parameters.

```
int params_server_ioctl(int fd, int dev_id, void *ioc_buf, int ioc_len);
```

- parameters
  - fd: the communication fd for ioctl and list.
  - dev_id: the dev_id for ioctl, for lustre ioctl, it should be unique value (as /dev/obd)
  - ioc_buf: ioc buf, same as original implementation.
  - ioc_len: the length of ioc_buffer.
- Return
  - $= 0$ success, $< 0$ error.
- Description
  - The API is used to handle ioctl request.

## 3.3 Proc tree on server

As discussed, a backend tree is maintained in the server, similar as procfs internally in linux, but platform independently. Then other module or obd will add/delete their proc parameters on the tree and the params server will locate the parameter entry by the tree.

### 3.3.1 proc tree structure

There will be an unique lustre_params_root (structure lustre_params_entry) for each server node. Each entry is allocated a name, value and correspondent read/write cb, as procfs in linux kernel. The structure is also similar as proc entry in linux kernel.

\begin{lstlisting}
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 list is NULL*/
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_lock lpe_rw_lock;
__u32 lpe_version;
void *lpe_data; /* The argument for the read and write callback */
int lpe_mode; /* dir, file or symbol_link*/
};
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);
\end{lstlisting}

And there are also several APIs associate with the tree. Since the lctl params will be used to set/get the parameters, instead of organizing all the parameters as procfs, so these API should be somewhat simple compared with real procfs API.

```
int lustre_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 lustre_params_delete_entry (struct lustre_params_entry *lpe, char *name);
struct lustre_params_entry * lustre_params_lookup_entry
(struct lustre_params_entry *lpe, char *name);
```

- parameters
  - lpe: the dir entry.
  - name: the name of the added/deleted/lookup entry.
  - read_cb: the read function.
  - write_cb: the write function.
  - data: the parameter put to the lpe_data.
- Return
  - Read, >= 0 the read length, < 0 error.
  - Write. = 0 success, < 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 proc tree.

```
int lustre_params_walk_through_entry (struct lustre_params_entry *lpe_root,
lustre_params_walk_cb_t *lpe_cb);
typedef_t int (lustre_params_walk_cb_t)(struct lustre_params_entry *lpe);
```

- parameters
  - lpe_root: the root of the proc tree.
  - lpe_cb: the callback for each entry.
- Return
  - Read, = 0 success, < 0 error.
- Description
  - The API walks through all the entry of the tree and call the callback function for each entry.

```
int lustre_params_ioctl(int dev_id, void *ioc_buf, int ioc_len);
```

- parameters
  - dev_id: for lustre ioctl, the dev_id is unique (as /dev/obd in kernel base)

- ioc_buf: the ioctl buffer.
- ioc_len: the length of ioctl buffer.

- Return
  - = 0 success, < 0 error

- Description
  - This API is used to handle ioctl request on server.

# 4 Use cases

1. Set/get/list proc parameters
   (a) Lctl set/get_params call the params_open to get the socket descriptor first. And on server side, it will setup the connection with the request and create another thread to handle the request.
   (b) It calls params_read/write, to pack the request and send to the server by the socket descriptor.
   (c) On server side, the handler thread handle the request by those API defined in 3.2.
2. Add/remove param entry
   (a) Obd or module call lprocfs code to add/delete the entry from the tree.
   (b) In lprocfs code, lustre_params_add/delete_entry will be called to add/delete entry of the tree.

# 5 Logic specification

## 5.1 lctl interface

### 5.1.1 open/close/read/write/list/close params

Current lctl implement set/get_params interface based on several posix system call open, read, write, glob, close, and all of them are based on local linux procfs. But in user base server, the "procfs" is maintained on server, and there are no local procfs at all, so those posix system calls are needed to be replaced by those lctl API defined in 3.1 (open–> params_open, read–>params_read, write–>params_write, glob–>params_list).

- params_open should connect the proc server with defined port and protocal (dev_path parameter), then on the proc server, the accepting thread creates a handling thread to handle the following request from the lctl client.

- params_close will send the close request to proc server, then proc server will determinate the handling thread, and client will close the socket descriptor.
- params_read/write APIs just needs pack requests(parameters) and send requests to proc server thread.
- params_list API will retrieve proc entries from the proc server by a connected socket stream. Sometimes the path pattern might be provided, so on proc server side, some reg expression lib needs to be used to choose the right entries matched the path_pattern.
- params_ioctl is similar as params_read/write, packing the parameters to proc server, on the server side, it will unpack the request and call obd_class_ioctl directly.

## 5.2 Server proc handler

On proc server side, there is a socket accept thread to receive the params/ioctl request from client, and put the request to the list. Then it will create a thread to handle the request. Current three are five requests(close, read, write, list, ioctl) need to be handled. For close, the handler thread will terminate itself. For ioctl, the handler will unpack the request and call obd_class_ioctl directly. For read/write/list, all of them will be based on the "backend" params tree.

### 5.2.1 params tree

In current implementation, all the module call lprocfs API to operate the procfs tree. The lprocfs api is implemented by exported linux procfs API, so it will only be built with linux kernel currently. And lprocfs code will still be used in the user base proc tree, but those linux kernel exported API and structure needed to replaced with our own structure and API.

Compared with procfs, the params tree is simple, and it only need provides 4 APIs, add_entry, remove_entry, lookup_entry, list_entry. And the tree will be protected by a global read/write lock, so when lookup or listing the entries, the tree will not be modified.

1. Add_entry

\begin{lstlisting}
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 */
/* lock the whole tree */

```
    lustre_params_write_lock(params_tree_lock);
    /* link the lpe_child to the lpe children list */
    lpe_child->lpe_next = lpe->lpe_subdir;
    lpe_child->lpe_parent = lpe;
    lpe->lpe_subdir = lpe_child;
    /* increase the refcount of the parent
    lustre_params_write_unlock(params_tree_lock);
    return 0;
    }
    \end{lstlisting}
```

1. remove_entry

```
\begin{lstlisting}
    static struct lustre_params_entry * lookup_entry (struct lustre_params_entry
*parent, char *name, int length);
    {
    for(entry = parent->lpe_subdir; entry ; entry = parent->lpe_next ) {
    if (!strncmp(entry->lpe_name, name, length))
    return entry;
    }
    return NULL;
    }
    struct lustre_params_entry * params_remove_entry (struct lustre_params_entry
*lpe, char *name)
    {
    /* lock the whole tree */
    lustre_params_write_lock(params_tree_lock);
    /* Remove the lpe_child to the lpe children list */
    for(entry = parent->lpe_subdir; entry ; entry = parent->lpe_next ) {
    pre = entry;
    if (strncmp(entry->lpe_name, name, strlen(name)))
    continue;
    pre->lpe_next = entry->next;
    /* decrease the refcount of the parent */
    }
    lustre_params_write_unlock(params_tree_lock);
    return 0;
    }
    \end{lstlisting}
```

1. lookup_entry (similar as link_walk_path process in linux kernel)

```
\begin{lstlisting}
    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;
lustre_params_read_lock(params_tree_lock);
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;
child = lookup_entry(parent, lookup_name, lookup_name_length);
if (child == NULL)
break;
if (IS_Symblol(child)) {
child = lookup_entry(parent, lookup_name, lookup_name_length);
}
/* If it is a symbol link, read name from the lpe_data and lookup again
if (last_component)
break;
else
parent = child;
lustre_params_read_unlock(params_tree_lock);
}
return child;
}
\end{lstlisting}
```

1. list_entry

```
\begin{lstlisting}
    static int match ( char *path_pattern, int path_pattern_len, char *path_name,
int name_len)
    {
    /* check whether the name is matched with path_pattern */
    /* Use regcompile to format the name */
    /* Use regexec to check whether the name is matched with path_pattern */
    /* matched return 1, other_wise return 0.
    }
    /* The stack will be used to help walk through all the entries */
```

```
int params_list_entry (struct lustre_params_entry *root, char *path_pattern,
params_list_cb_t *callback_t)
{
/*Got the name from each entry */
struct lustre_params_entry *parent;
struct lustre_params_entry *child;
char *lookup_name, *match_string;
int lookup_name_length = 0;
parent = &lustre_params_root_entry; /*initialize the root entry */
match_string = name = path;
lustre_params_read_lock(params_tree_lock);
/* Use stack to help listing the entries */
entry = root;
push_stack (stack, entry);
while (stack_not_empty(stack)) {
if (entry->lpe_subdir) {
/* Push to the stack */
push_stack(entry);
entry = entry->lpe_subdir;
} else {
/* handle the leaves of the tree*/
callback (entry);
for(entry = entry->lpe_next; entry; entry = entry->lpe_next )
callback (entry);
/* pop the entry and move to next entry */
pop_stack(entry);
entry = entry->lpe_next;
}
}
lustre_params_read_unlock(params_tree_lock);
return 0;
}
\end{lstlisting}
```

### 5.2.2 The server proc/ioctl handler

The handler will handle three request, read, write and list. For read/write, it will locate the entry first by lustre_params_lookup_entry, then get/set the entry by the read/write callback function attached by the entry.

```
\begin{lstlisting}
int params_server_read(int fd, char *path, int path_len, int offset)
{
obd_alloc(buf);
entry = params_lookup_entry (path);
entry->read_cb(entry, buf);
/* write the buffer to fd */
```

```
}
\end{lstlisting}
For list:
\begin{lstlisting}
int params_server_list(int fd, char *path_pattern, int path_len, int offset)
{
obd_alloc(buf);
/* locate the first the name which is not path_pattern,
* for example, for ldlm.namespaces.*mdc*.lock_count, ldlm.namespaces is
* located first. then the left of the path_pattern will be put to list API
*/
entry = params_list_entry (left, list_cb);
}
\end{lstlisting}
```

# 6 State Management

As discussed, a global read/write lock will protect the tree being modified when lookup or list the entries.

# 7 Alternatives

Another alternative way for communication between lctl client and proc server