Understanding the Proc File System

Author: Ravi Kiran UVS

Introduction

Proc is an in-memory file system used to provide file based communication with the kernel. It allows registration/unregistration functions to create files and remove files on the fly. Any kernel component that wants to communicate with the user can create a file under the proc and that can be used to exchange data. Otherwise, it will have to create a system call which the user has to know or has to use some sort of IPC. Reading from and writing to a file is a common operation and hence it makes lot of sense to have the communication channel as a file. Proc create an infrastructure for doing that and creates a simpler interface for kernel components to register files and the corresponding handler (these handlers will provide the actual functionality).
All the files under the proc file system reside in memory. There are stored only in the RAM and hence they will be destroyed when the kernel reboots.

Proc Infrastructure

Each proc entry(file and directory) is described using a structure proc_dir_entry.

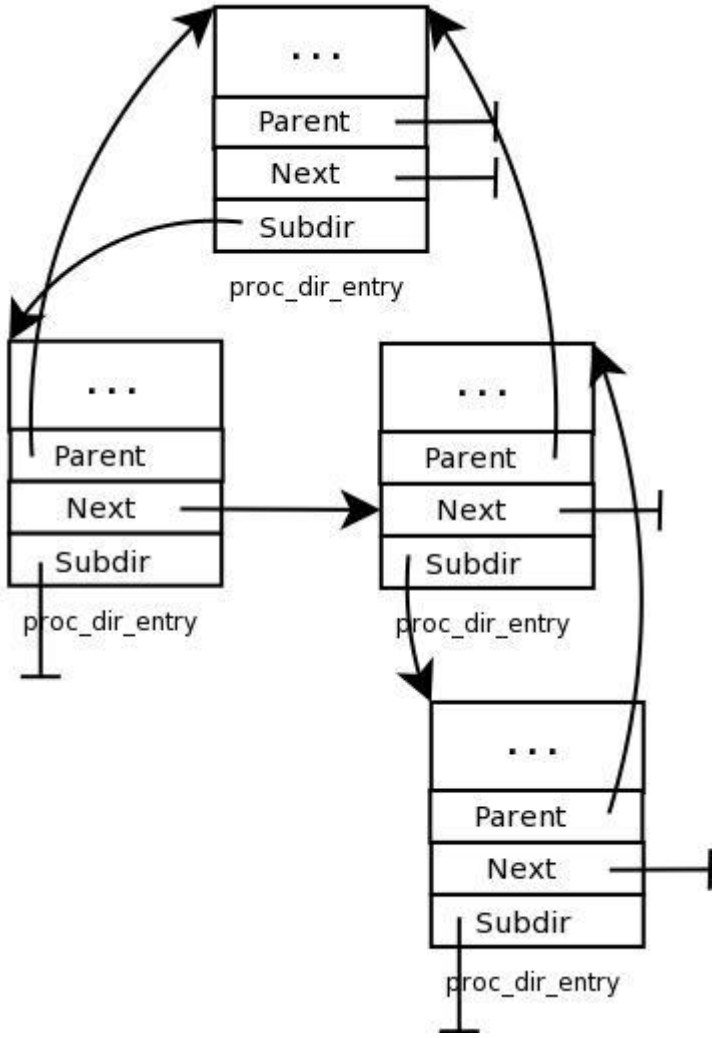
```
struct proc_dir_entry {
    unsigned short low_ino;
    unsigned short namelen;
    const char *name;
    mode_t mode;
    nlink_t nlink;
    uid_t uid;
    gid_t gid;
    unsigned long size;
    struct inode_operations * proc_iops;
    struct file_operations * proc_fops;
    get_info_t *get_info;
    struct module *owner;
    struct proc_dir_entry *next, *parent, *subdir;
    void *data;
    read_proc_t *read_proc;
    write_proc_t *write_proc;
    atomic_t count;         /* use count */
    int deleted;        /* delete flag */
    kdev_t    rdev;
};
```

The important fields are

name	Name of the file
uid	Owner ID of the file
gid	Owner group ID of the file
size	Size of the file
proc_iops	Inode operations table to be set on the proc inode

proc_fops	File operations table to be set on the proc inode
data	Used to store extra data. Proc uses it to store the target path in case of a link.
get_info	Older handler to read from the proc file
read_proc	Read handler of the proc file. Called when something has to be read from the proc file.
write_proc	Write handler of the proc file. Called when something has to be written into the proc file.
parent	Points to the parent entry. For root most entry, this is null.
next	Points to the next entry. This list is the list of all entries in a directory. Every entry points to the same parent.
subdir	Points to the first child. For leaf entries, this is null.

This parent, next and subdir fields are used to store the hierarchy of the proc entries. Each proc entry stores the get_info, read_proc and write_proc handlers. get_info and read_proc handlers are used for handling read requests whereas the write_proc handler is is used to handle write requests.



New entry creation

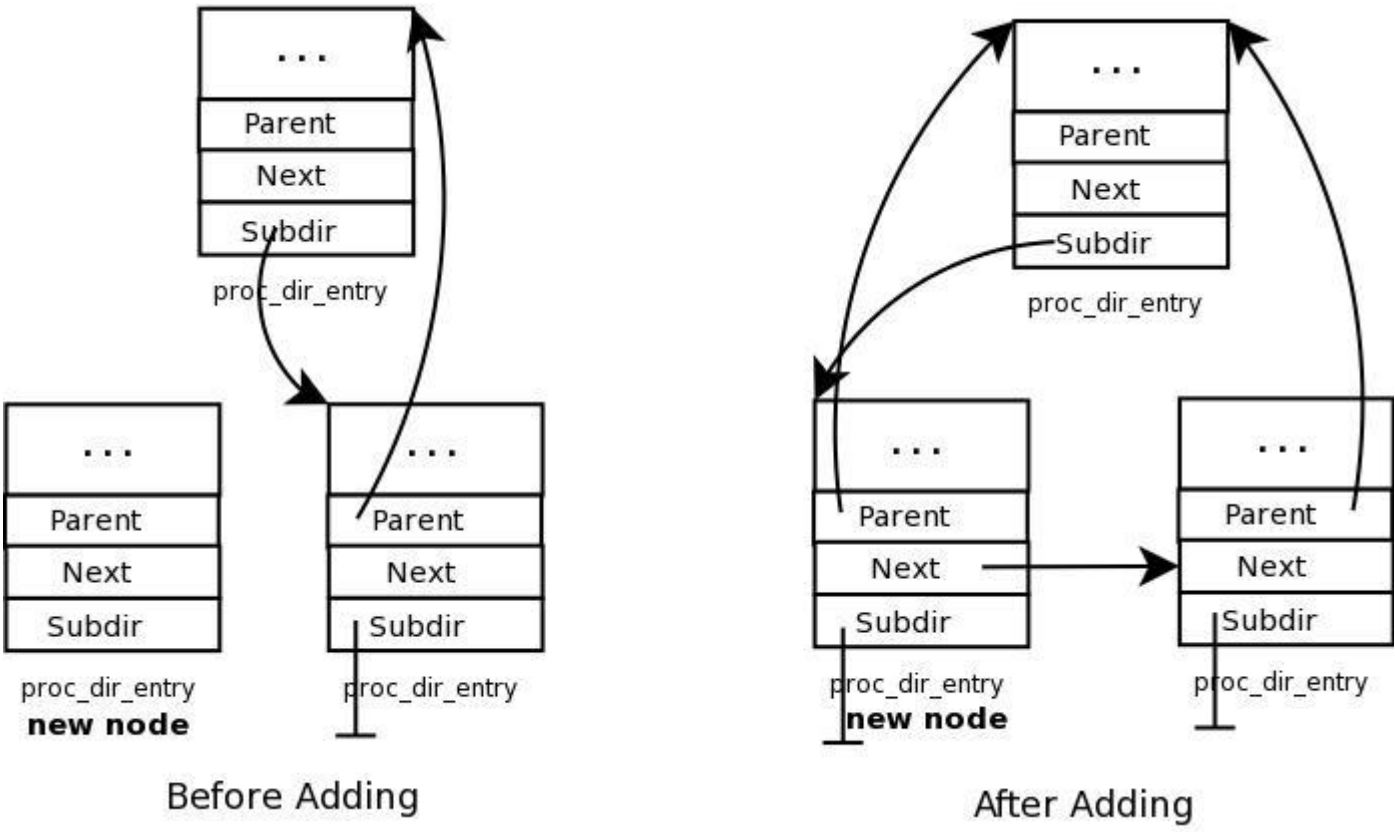
New entries are created under proc using the call create_proc_entry. The prototype looks like this `struct proc_dir_entry *create_proc_entry(const char *name, mode_t mode, struct proc_dir_entry *parent);`

This call takes a name, mode and the parent entry and creates an entry under the parent with the name as 'name' and fills the important fields like proc_iops, prof_fops etc on the entry. This will also insert the entry in the proper location in the proc hierarchy. The newly created proc entry will be returned. The caller will have to fill the remaining fields like size, get_info, read_proc, write_proc, uid, gid etc.

The name can also be a relative path under the proc. For example, if the name is 'proctest' a file will be created which will look like '/proc/proctest'. If the name is 'fs/proctest', the file will be created as '/proc/fs/proctest'.

This entry will be inserted at the beginning of the subdir list of the parent entry. The root most entry of the proc file system is defined as a global variable proc_root. This can be used to pass as the parent for this function. For example, if we want to create a file /proc/proctest, we can call like this

```
proc_dir_entry *new_entry = create_proc_entry("proctest", S_IFREG,
&proc_root);
```



If the entry is a directory, it sets the proc_fops with the table with lookup handler (lookup handler is needed only for directories). If it is a link, it sets to a different inode operations table which has readlink and follow_link handlers.

For regular files, it sets the proc_fops with the table with llseek, read and write handlers (these are the proc handlers). For directories, it sets it to a table with only one operation readdir. The will be called when the directory contents have to be read.

These are set by default but the caller can always overwrite them after creating the entry. In that case, the handlers of the new tables will be called and the caller is responsible to handle them. This strategy is adopted by files which copy lot of content. Proc files handling through the read_proc callback has to copy into a proc buffer which will be copied into the user space. Instead of this, we can register a new file operations table so that our handlers get called. We can directly read/write into the user buffer. The handler for '/proc/ksyms' uses this strategy.

Entry deletion
The function to be used to delete an entry is `void remove_proc_entry(const char *name, struct proc_dir_entry *parent);`

VFS interaction

Proc registers proc file system with the kernel. The mount point is chosen as "/proc". The file system name is "proc". Proc has to support directory browsing, file access. So, it supports

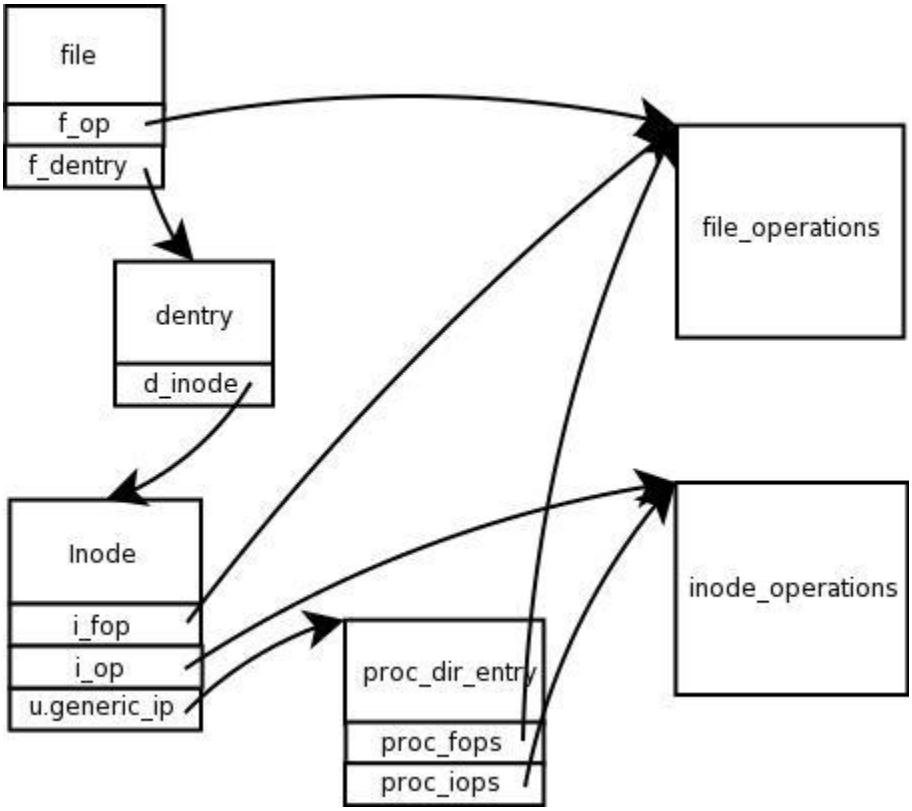
lookup handler of the inode operations table, readdir, read, write and llseek of the file operations table.
The read_super handler of the file system type structure is called when the file system is mounted. This will fill the super block structure with the proper fields. The important field is the s_root which should be filled with a dentry object containing the inode object for the root most directory of the file system.

Lookup operations
The lookup handler of the inode operations is the place where new inode objects (wrapped in dentries) are generated. The root most entry is returned at the time of mounting. For every inode, proc has to set two important fields i_op and i_fop i.e., the inode operations and the file operations tables. The lookup handler is part of the inode operations. proc_lookup is the lookup handler for proc.
For every inode returned, a pointer to the corresponding proc_dir_entry is stored in the u.generic_ip (it is of type void *). With this, we can easily get the proc entry structure from the inode.

```
struct dentry *proc_lookup(struct inode * dir, struct dentry *dentry)
```

When the kernel looks up for some file, it calls the lookup handler of the inode operations table of the parent's inode. When proc_lookup is called, the inode of the parent entry is passed and also the dentry of the new entry which has to be filled with the new inode. Since the proc entry pointer is stored in the inode object, we can get the proc entry of the parent inode. The hierarchy is already present. So, the children of the parent entry are scanned to see if there is any entry with the matching name (the required name is stored in the dentry argument). If it finds an entry, it allocates an inode, sets the proc entry in the u.generic_ip of the inode and fills the required inode fields from the proc entry structure. It sets the i_op and i_fop fields of the inode with the proc_iops and proc_fops fields of the entry respectively.

This callback is very important because this is the place to set the pointers to the operation tables. The following diagram shows the relation between various data structures after a file object is constructed for the proc entry.



readdir operation

When the user wants to see the contents of a directory, the readdir handler of the file operations table of the directory is called. proc_readdir is registered as the handler. This is its prototype:

```
int proc_readdir(struct file * filp,void * dirent, filldir_t filldir);
```

From the filp->f_dentry->d_inode, we can get the inode object. The proc entry is stored in the inode->u.generic_ip. So, we can get the proc entry structure from the file object. The subdir field of the proc entry points to its children.

```
int proc_readdir(struct file * filp,
    void * dirent, filldir_t filldir)
{
    struct proc_dir_entry * de;
    unsigned int ino;
    int i;
    struct inode *inode = filp->f_dentry->d_inode;

    ino = inode->i_ino;
    de = (struct proc_dir_entry *) inode->u.generic_ip;
    if (!de)
        return -EINVAL;
    i = filp->f_pos;
    switch (i) {
        case 0:
            if (filldir(dirent, ".", 1, i, ino, DT_DIR) < 0)
                return 0;
            i++;
            filp->f_pos++;
            /* fall through */
        case 1:
            if (filldir(dirent, "..", 2, i,
```

```
                    filp->f_dentry->d_parent->d_inode->i_ino,
                    DT_DIR) < 0)
                return 0;
            i++;
            filp->f_pos++;
            /* fall through */
        default:
            de = de->subdir;
            i -= 2;
            for (;;) {
                if (!de)
                    return 1;
                if (!i)
                    break;
                de = de->next;
                i--;
            }

            do {
                if (filldir(dirent, de->name, de->namelen, filp->f_pos,
                        de->low_ino, de->mode >> 12) < 0)
                    return 0;
                filp->f_pos++;
                de = de->next;
            } while (de);
    }
    return 1;
}
```

Write operation

The handler for write operation registered by proc is proc_file_write. The prototype is this funciton is:

```
ssize_t proc_file_write(struct file * file, const char * buffer, size_t
count, loff_t *ppos)
```

The proc entry can be easily obtained from the file pointer. If the proc entry has a write_proc handler registered, it will be called. Otherwise it returns a -EIO.

Read operation

The handler for read operation registered by proc is proc_file_read. The prototype of this function is:

ssize_t proc_file_read(struct file * file, char * buf, size_t nbytes, loff_t *ppos);

The handler implementation is slightly complex. If the proc entry has get_info handler, it calls get_info. Otherwise it checks for read_proc, and if it is available, it will be called. So, before calling the handler, a page of memory will be allocated and this will be used to copy data from the handler to the user space. The read_proc/get_info handlers write into the page and this will be copied into the user space.

```
/* buffer size is one page but our output routines use some slack for
overruns */
#define PROC_BLOCK_SIZE    (PAGE_SIZE - 1024)

static ssize_t
proc_file_read(struct file * file, char * buf, size_t nbytes, loff_t *ppos)
{
    struct inode * inode = file->f_dentry->d_inode;
    char     *page;
```

```
    ssize_t    retval=0;
    int    eof=0;
    ssize_t    n, count;
    char    *start;
    struct proc_dir_entry * dp;

    dp = (struct proc_dir_entry *) inode->u.generic_ip;
    if (!(page = (char*) __get_free_page(GFP_KERNEL)))
        return -ENOMEM;

    while ((nbytes > 0) && !eof)
    {
        count = MIN(PROC_BLOCK_SIZE, nbytes);

        start = NULL;
        if (dp->get_info) {
            /*
             * Handle backwards compatibility with the old net
             * routines.
             */
            n = dp->get_info(page, &start, *ppos, count);
            if (n < count)
                eof = 1;
        } else if (dp->read_proc) {
            n = dp->read_proc(page, &start, *ppos,
                      count, &eof, dp->data);
        } else
            break;

        if (!start) {
            /*
             * For proc files that are less than 4k
             */
            start = page + *ppos;
            n -= *ppos;
            if (n <= 0)
                break;
            if (n > count)
                n = count;
        }
        if (n == 0)
            break;    /* End of file */
        if (n < 0) {
            if (retval == 0)
                retval = n;
            break;
        }

        /* This is a hack to allow mangling of file pos independent
          * of actual bytes read.  Simply place the data at page,
          * return the bytes, and set `start' to the desired offset
          * as an unsigned int. - Paul.Russell@rustcorp.com.au
         */
         n -= copy_to_user(buf, start < page ? page : start, n);
        if (n == 0) {
            if (retval == 0)
                retval = -EFAULT;
            break;
        }
```

```
        *ppos += start < page ? (long)start : n; /* Move down the file */
        nbytes -= n;
        buf += n;
        retval += n;
    }
    free_page((unsigned long) page);
    return retval;
}
```