Debugging the Linux Kernel with debugfs

http://opensourceforu.com/2010/10/debugging-linux-kernel-with-debugfs/
http://blog.techveda.org/debugfs/

Introduced by Greg Kroah-Hartman in December 2004.

Debugfs is a simple memory-based filesystem, designed specifically to debug Linux kernel code, and not to be confused with the debugfs filesystem utility.

Debugfs helps kernel developers **export large amounts of debug data into user-space.**

It has no rules about any information that can be put in, unlike `/proc` and `/sysfs`, and hence is encouraged for use in debugging. Using `printk` can suffice when reading values — but it does not allow developers to change values of variables from user-space.

Setting up DebugFS

If you are using one of the latest distributions, chances are that debugfs is already set up on your machine. If you're compiling the kernel from scratch, make sure you **enable debugfs in the kernel configuration**. Once you reboot to your newly compiled kernel, check if debugfs is already mounted, with the following command:

```
# mount | grep debugfs
none on /sys/kernel/debug type debugfs (rw)
```

If you see output as above, you have debugfs pre-mounted. If not, you can mount it (as root) with the command shown below:

```
# mount -t debugfs nodev /sys/kernel/debug
```

If you want it to be available on every reboot, append an entry in `/etc/fstab` as follows:

```
debugfs /sys/kernel/debug debugfs defaults 0 0
```

Once mounted, you can view a lot of files and directories in `/sys/kernel/debug`, each belonging to one or the other subsystem.

The debugfs API

To access the API, include `linux/debugfs.h` in your source file. To be able to use debugfs, we start by creating a directory within `/sys/kernel/debug`, which is an ideal way to start. The rest of the files can be placed within this directory.

```
struct dentry *debugfs_create_dir(const char *name, struct dentry
*parent);
```

Here, is the name of the directory, and parent is the parent directory (if null, the directory is created in `/sys/kernel/debug`). If debugfs is not enabled in the kernel, `ENODEV` is returned.

If you need to create a single file within debugfs, you can call the following function:

Here, **`name`** is the file name to be created;

mode stands for the permissions of the created file;

`parent` specifies the parent directory in which the file should be created (if null, it defaults to the debugfs root, `/sys/kernel/debug`); `data` is the type `inode.i_private` and `fops` is the file operations.

If you need to write to and read from a single value, you can use this to create an unsigned 8-bit value:

```
struct dentry *debugfs_create_u8(const char *name, mode_t mode,
struct dentry *parent, u8 *value);
```

Here, `value` is a pointer to a variable that needs to be read and written to.

A few other helper functions to create files with single integer values are:

- `struct dentry *debugfs_create_u16`
- `struct dentry *debugfs_create_u32`
- `struct dentry *debugfs_create_u64`

(Refer to `fs/debugfs/file.c` for more information.)

Similar functions that give hex output, are:

```
dentry *debugfs_create_x8(const char *name, mode_t mode, struct
dentry *parent, u8 *value)
dentry *debugfs_create_x16
dentry *debugfs_create_x32
^^dentry *debugfs_create_x64
```

Note: `debugfs_create_x64` is the most recent edition of the API; added in May 2010 by Huang Ying, this tells us that debugfs is still in active development.

Clean up!

One important thing to remember is that all the files and directories created using the above APIs are to be cleaned up by the creator, using the following code:

```
void debugfs_remove(struct dentry *dentry)
```

Here, `dentry` is a pointer to the file or directory that needs to be cleaned up.

Other available DebugFS APIs are:

- `debugfs_create_symlink` — create symbolic link in debugfs
- `debugfs_remove_recursive` — recursive removal of a directory tree
- `debugfs_rename` — rename a file or directory

- `debugfs_initialized` — to know whether debugfs is registered
- `debugfs_create_bool` — to read/write a boolean value
- `debugFS_create_blob` — to read a binary large object

You can refer to code for these and other functions in `fs/debugfs/{inode.c, file.c}` in the kernel directory.

On a general note, debugfs is not considered to be a stable Application Binary Interface (ABI), and hence it should only be used to collect debug information on a temporary basis. For more information in this regard, you can refer to an LWN article.

Code sample

Note: You need to be running the latest kernel to make this program work.

This sample program, `my_debugfs.c`, shows a use of debugfs.

struct dentry *debugfs_create_file(const char *name, mode_t mode, struct dentry *parent, void *data, struct file_operations *fops);

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/debugfs.h> /* this is for DebugFS libraries */
#include <linux/fs.h>
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Surya Prabhakar <surya_prabhakar@dell.com>");
MODULE_DESCRIPTION("sample code for DebugFS functionality");
#define len 200
u64 intvalue,hexvalue;
struct dentry *dirret,*fileret,*u64int,*u64hex;
char ker_buf[len];
int filevalue;
/* read file operation */
static ssize_t myreader(struct file *fp, char __user *user_buffer,
                                size_t count, loff_t *position)
{
     return simple_read_from_buffer(user_buffer, count, position,
ker_buf, len);
}

/* write file operation */
static ssize_t mywriter(struct file *fp, const char __user
*user_buffer,
                                size_t count, loff_t *position)
{
        if(count > len )
                return -EINVAL;
        return simple_write_to_buffer(ker_buf, len, position,
user_buffer, count);
}

static const struct file_operations fops_debug = {
        .read = myreader,
        .write = mywriter,
};

static int __init init_debug(void)
{
    /* create a directory by the name dell in /sys/kernel/debugfs
*/
    dirret = debugfs_create_dir("dell", NULL);

    /* create a file in the above directory
    This requires read and write file operations */
    fileret = debugfs_create_file("text", 0644, dirret,
&filevalue, &fops_debug);

    /* create a file which takes in a int(64) value */
    u64int = debugfs_create_u64("number", 0644, dirret,
&intvalue);
    if (!u64int) {
        printk("error creating int file");
        return (-ENODEV);
    }
    /* takes a hex decimal value */
    u64hex = debugfs_create_x64("hexnum", 0644, dirret,
&hexvalue );
    if (!u64hex) {
        printk("error creating hex file");
        return (-ENODEV);
    }

    return (0);
}
module_init(init_debug);

static void __exit exit_debug(void)
{
    /* removing the directory recursively which
    in turn cleans all the file */
    debugfs_remove_recursive(dirret);
}
module_exit(exit_debug);
```