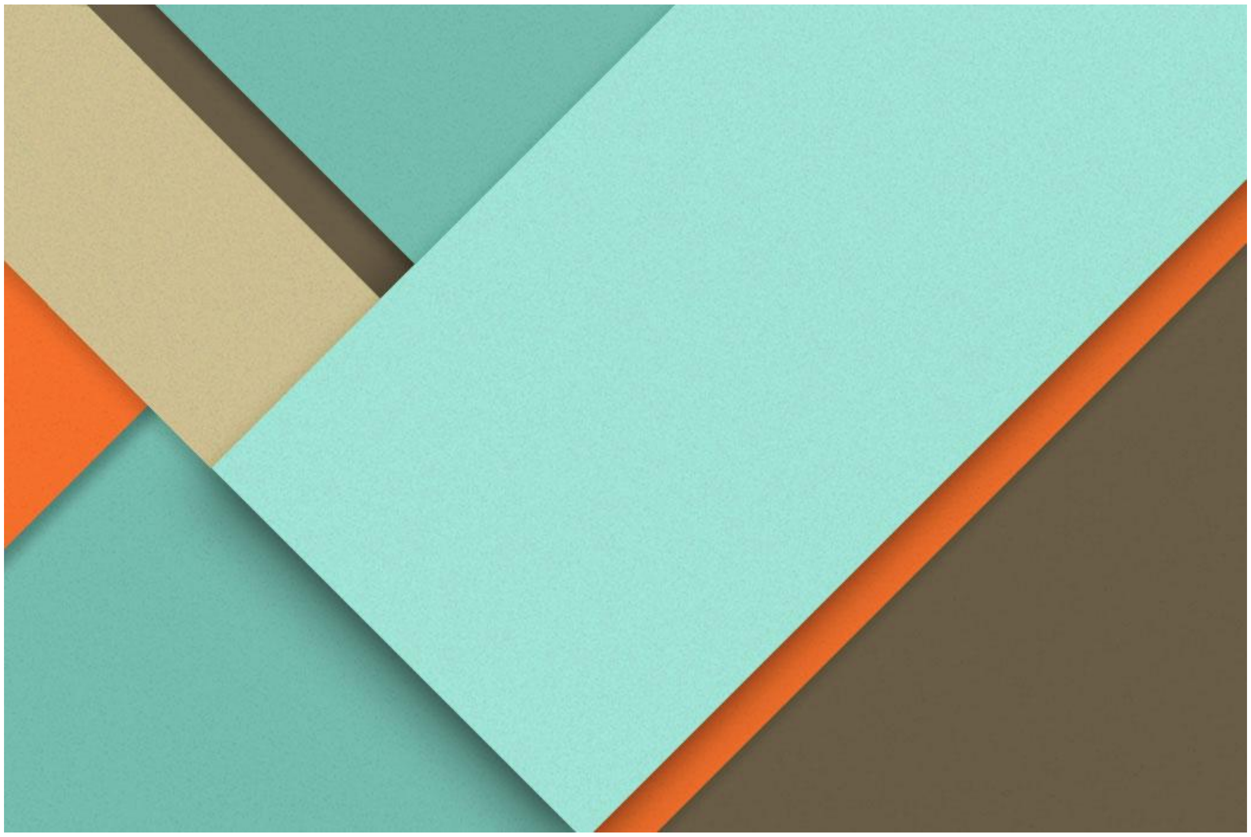

Exploring DebugFS

Reading Elective Report under Prof. Badrinath R.

—

Divyam Agrawal (IMT2019028)

Nandakishore S Menon (IMT2019057)

Rachna Kedigehalli (IMT2019069)

IIIT Bangalore

Overview

The project aimed to explore the potential use cases of DebugFS as a tool for helping developers with debugging their code. The project involved an in-depth investigation of DebugFS, including its features, limitations, and potential use cases. As part of the project, DebugFS was integrated into EROFS as an example of how it could be used to help developers.

Goals

1. Investigate the features and potential applications of DebugFS.
2. Integrate DebugFS into EROFS as an example.
3. Learn more about the Linux kernel and how to compile the kernel from scratch.
4. Understand the process of mounting drivers in Linux.
5. Document how to integrate DebugFS into a new module, compile the kernel, and mount drivers for new developers.

Milestones

I. Integration of DebugFS in a Dummy Module

We built a dummy module and integrated DebugFS into it. We created a makefile for the module and learned how to compile it. Using DebugFS, we explored how to access and manipulate kernel data structures and variables from user space.

II. Identification of a suitable Module for DebugFS Integration

We identified EROFS as a suitable file system for integrating DebugFS. We determined the features and requirements of EROFS and explored how DebugFS could be used to help developers working with this file system.

III. Exploration of EROFS code

We began reading the EROFS code to identify potential integration points for DebugFS. We analyzed the code to determine areas where DebugFS could be used to provide additional information to the developers.

IV. Learning to Compile Linux Kernel from Scratch

We learned how to compile the Linux kernel from scratch. We explored the different build options and configurations available and gained a better understanding of the kernel compilation process.

V. Modularization of DebugFS code and added more functionalities

We modularized the DebugFS code inside EROFS to provide a more robust and comprehensive debugging tool for developers working with EROFS. We developed additional functionalities using DebugFS to help developers working with EROFS.

Descriptions

What is DebugFS?

DebugFS is a virtual file system that allows developers to quickly access and modify kernel data structures and variables from within user space. It was specially designed for debugging purposes.

DebugFS can also be used to enable kernel code tracing and profiling, which is especially beneficial for finding performance bottlenecks and other issues.

Unlike /proc, which is only meant for information about a process, or sysfs, which has strict one-value-per-file rules, DebugFS has no rules at all. Developers can put any information they want there.

To compile a Linux kernel with the DebugFS, the CONFIG_DEBUG_FS option must be set to yes.

It is typically mounted at /sys/kernel/debug with a command such as:

```
mount -t debugfs none /sys/kernel/debug
```

What is EROFS?

EROFS (Enhanced Read-Only File System) is a read-only file system designed for embedded systems and mobile devices. It is designed for high storage efficiency and quick data access, making it excellent for usage in devices with limited storage space.

EROFS uses a unique compression algorithm that reduces the size of files and directories, resulting in lower storage usage. It also contains advanced capabilities like file-level deduplication, which decreases storage requirements even further.

Approach

Build Linux Kernel from Scratch

We followed [1] (G Jetvic, 2020) to build the Linux kernel from scratch.

Here are the steps we followed -

1. Download the Source Code

   Visit the official kernel website and download the latest kernel version.

2. Extract the Source Code

```
tar xvf linux-<version>.tar.xz
```

3. Install Required Packages

```
sudo apt-get install git fakeroot build-essential ncurses-dev xz-utils
libssl-dev bc flex libelf-dev bison
```

4. Configure Kernel

```
cd linux-<version>
cp -v /boot/config-$(uname -r) .config
```

To make changes to the configuration file-

```
make menuconfig
```

The configuration menu opens up where we can select different options.

Exit the menu since we don't need to change any default settings for this project.

5. Build the Kernel

```
make
```

Note the process may require a significant amount of time to complete, depending on the system speed.

If you are compiling the kernel on Ubuntu, you may receive the following error that interrupts the building process:

"No rule to make target 'debian/canonical-certs.pem"

Disable the conflicting security certificates by executing the two commands below:

```
scripts/config --disable SYSTEM_TRUSTED_KEYS
scripts/config --disable SYSTEM_REVOCATION_KEYS
```

Start the building process again with **make**, and press **Enter** repeatedly to confirm the default options for the generation of new certificates

Install the required modules with this command:

```
sudo make modules_install
```

Finally, install the kernel with this command:

```
sudo make install
```

6. Reboot and Verify Kernel Version

When the above steps are completed, reboot the machine.

When the system boots up, verify the kernel version using the command:

```
uname -mrs
```

The terminal prints out the current Linux kernel version.

Identifying functions within EROFS and adding DebugFS

We found the EROFS module within */fs/erofs* directory. Being a Read-only filesystem, we aimed at logging information when the inode table is instantiated and read-from and when a read operation is performed.

DebugFS requires the following variables, which have been packaged by us into a structure `debugfs_for_erofs`:

- `ker_buff`: A char array into which data is written, which debugFS interfaces as a file in a directory we declare within /sys/kernel/debug
- `len`: Length of ker_buff
- `pos`: the next free index in `ker_buff`

- `dir`: the dentry struct for the directory within `/sys/kernel/debug`
- `erofs_add_debug_info`: function pointer to a custom function that writes content to the ker_buff and gets called for logging

DebugFS also needs the `file_operations` structure to be populated with a reader and writer function.

```
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,
      re_debugfs_for_erofs.ker_buf, re_debugfs_for_erofs.len);
}



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

First, we need to define the debugfs file for erofs and the directory within /sys/kernel/debug where the file will reside. This must be done when the EROFS module is intialized, which we found within the erofs_module_init of the *super.c* file.

```
struct debugfs_for_erofs re_debugfs_for_erofs;
re_debugfs_for_erofs.dir = debugfs_create_dir("erofs_super", 0);
struct dentry *fileret = debugfs_create_file("read_iter", 0644,
re_debugfs_for_erofs.dir , &filevalue, &fops_debug)
```

The above code placed within the `erofs_module_init` would create the debugFS file in `/sys/kernel/erofs_super`.

The `erofs_add_debug_info` is set to the function given below and can be called in methods where loggin would be required.

```
void erofs_add_debug_info(const char *info, int len, char ker_buf[])
{
      if(strlen(ker_buf) + strlen(info) + 1 < len ) {
            strcat(ker_buf, info);
      }
      else {
            ker_buf[0] = ker_buf[1] = '.'; ker_buf[2]='\0';

            strcat(ker_buf,info);
      }

}
```

`The function can be called as given below:`

```
re_debugfs_for_erofs.erofs_add_debug_info("Read Superblock\n",
re_debugfs_for_erofs.len, re_debugfs_for_erofs.ker_buf);
```

We also timed the execution of certain functions like `erofs_init_fs_context` by using `ktime_get_ns function.`

Checking debugFS entries and triggering logs

The `erofs_add_debug_info` have been added to function responsible for reading super block, populating the inode table, reading from the inode table, caching inode entries etc.

First we create an EROFS image and mount it.

```
mkfs -t erofs erofsfolder /usr/share/man

mkdir /mnt/eromount

mount -t erofs erofsfolder /mnt/eromount/
```

After mounting the image, we can check the logs by using:

```
cat /sys/kernel/debug/erofs_super/read_iter
```

Then we can navigate to /mnt/eromount and do an `ls` command and again check the debugFS entry. We node that before doing an ls, EROFS populated the inode table. Upon doing an ls, EROFS reads the inode tables and caches it. Doing an ls again and checking the logs reveal that the action results in a cache read.

Thus debugFS can reveal the functioning of kernel modules and help debug it while developing it.

What next?

The next step would be to customize the read and write functions to create custom commands that could be sent as part of the string passed to `erofs_add_debug_info`. This would be done by checking the string passed within the writer and selectively performing actions based on its contents.

Another important addition to the utility would be to secure the ker_buff with a semaphore, such that multiple users can simultaneously use the utility without running into unwanted errors.

We attempted to do the same, however, our Ubuntu installation started misbehaving unexpectedly and were not able to finish it.

Customizing the writer function would give a lot of power to the developers, to create logs specific to their use cases and would be a useful addition to this debugFS utility.

References

[1] G. Jevtic, “How to build linux kernel {step-by-step},” *Knowledge Base by phoenixNAP*, Nov. 12, 2020. https://phoenixnap.com/kb/build-linux-kernel (accessed May 04, 2023).