Unit tests for Linux kernel hackers

How to avoid hours of painful debugging

Mihir Mehta

Systems Core Group
Samsung Research Institute - Noida

mihir.mehta@samsung.com

February 3, 2014

Overview

Overview of the material.

- What are unit tests?
- Why unit tests?
- Accessing unit test results
- Summary

What are unit tests?

What are unit tests?

- Ideally, a unit test involves calling a *single* function and *verifying* its return value against an *expected* value.
- Running a test can have two outcomes.
  1. Pass: Congrats! Your function works correctly on this test input!
  2. Fail: This again raises multiple outcomes:
     - 2.1 Function body is wrong. (bad, but manageable if you know what you're doing.)
     - 2.2 Expected value is wrong (indicates you don't know what you're doing, i. e. very bad.)

Why unit tests?

- Central component of test-driven design.
- Provide convenient way to verify our code.
- Save manual labour (i.e. less time at desk!)
- Crucial: help prevent regressions.

Accessing unit test results

- In a userspace program, (i.e. what we're used to) it's simple to make a separate test executable and link the functions we are testing into this executable.
- This way, we maintain a logical separation of the test code from the codebase itself.
- We also avoid bloating our executables.
- Making a separate test executable is not possible within kernel code - why?

Accessing unit test results

- Making a separate test executable is not possible within kernel code - why?
    - Can't be a userspace program - userspace programs can't link to kernel code (privilege levels and other issues)
    - Can't be a kernel - how are you going to run two kernels at once?
- So, how do we implement unit tests for kernel code now?

Accessing unit test results

- So, how do we implement unit tests for kernel code now?
  - Kernel modules!
- With modules, we can easily access kernel functions and test them.
- We can run all the tests at once, in the init function of the module, after setting a load priority for the module.
- If we use loadable modules (instead of static modules), we can avoid kernel bloat too - simply unload the module when done.
- One detail remains - how do we access the results of these tests in userspace?

Accessing unit test results

- How do we access the results of our unit tests in userspace?
  - Special kernel filesystems - sysfs and debugfs
- It's actually quite common for kernel variables to need to be readable and possibly modifiable from userspace. For this purpose, the Linux kernel provides certain special file systems.
  - procfs - NOT to be used for debugging.
  - sysfs - should not be used for debugging (but we do it anyway).
  - debugfs - explicitly intended for debugging.
- sysfs - used by many modules which need to communicate with user space; also abused for debugging; has strict restrictions on the types of exported values (only one numeric value per file).
- debugfs - used by many modules for debugging; has no restrictions on the types of exported values.
- Both of the above are sufficiently widely used to appear in nearly all distribution kernels (including ours).
- sysfs is generally mounted at `/sys`, while debugfs is generally mounted at `/sys/kernel/debug`.

Using debugfs for debugging

- Every kernel module should store its debugging files in a separate directory in debugfs. Optionally, subdirectories can be used for better organisation.
- This function is used for creating a directory -

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

  If the value of parent is NULL, then the directory is created at the debugfs root.
- The struct dentry * pointer obtained from this function can now be used to create files in the directory, using this function -

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

Using debugfs for debugging

- Usually, though, the function `debugfs_create_file` is too bulky for our purposes - we'd rather not have to make a set of file operations.
- The debugfs module also provides a few convenience functions for common debugging use cases, to remove the need for making sets of file operations.
- In cases where we only want to export a numeric value, we can take advantage of these functions:
  - `struct dentry *debugfs_create_u8(const char *name, umode_t mode, struct dentry *parent, u8 *value);`
  - `struct dentry *debugfs_create_u16(const char *name, umode_t mode, struct dentry *parent, u16 *value);`
  - `struct dentry *debugfs_create_u32(const char *name, umode_t mode, struct dentry *parent, u32 *value);`
  - `struct dentry *debugfs_create_u64(const char *name, umode_t mode, struct dentry *parent, u64 *value);`

Acknowledgements

- `doc/Documentation/filesystems/debugfs.txt` in the kernel source. (primary reference, authored by Jonathan Corbet.)