Writing Better Function Tests with GCOV

Masami Hiramatsu
Who Am I

Masami Hiramatsu (Linaro)
- Tech Lead in Socionext Landing Team
- Maintainer for:
  - kprobes and dynamic tracing in Linux kernel
  - ftracetest (a part of kselftest, function tests for ftrace)
Agenda

- Function Tests
  - Usual Issues
- GCOV and LCOV
  - How to use in userspace
- GCOV Kernel
  - Subsystem profiling
- Writing Function Tests with GCOV
- Ftracetest
  - Improving ftracetest with GCOV
  - Typical Untested Patterns
  - Pitfalls
Function(al) Tests

Tests each “function(feature)” of software
- Not function-level unit test :)
- Not a stress test
- It is a kind of regression test

Goal of function test
- For ensuring the “function” works as we expected
- Make sure no regressions while upgrading
Function Tests in Linux Kernel

There are several function tests
- Boot-time self tests
- Test (sample) modules
- Runtime tests
- Test collection: kselftests
Usual Issues on Writing Tests

A bug was found!
   -> Why was not that tested?
Want to write a test!
   -> What functions are not tested?

We need a measurement / visualizing tool for writing tests
GCOV and LCOV

GCOV: Coverage measurement tool for GCC
- Shows which “Line of code” is executed
- Calculate the coverage rate per line for each file

LCOV: Gcov visualizing wrapper tool
- Analyze multiple files at once
- Visualize the report in HTML
  - Show per-line and per-function coverage rate
  - Source-code based coverage report
LCOV Examples

Overview

LCOV - code coverage report

Current view: top level
Test: gcov.info
Date: 2018-09-04 10:11:36

Lines: 12013 / 17528, 68.5%
Functions: 1302 / 1881, 69.2%

Directory | Line Coverage | Functions
-----------|---------------|-------------
arch/x86/include/asm | 69.1% / 94 / 136 | 100.0% / 5 / 5
include/asem-generic | 80.0% / 8 / 10 | - / 0 / 0
include/linux | 78.9% / 232 / 294 | 77.8% / 7 / 9
include/linux/sched | 86.7% / 13 / 15 | 100.0% / 1 / 1
include/linux/unaligned | 100.0% / 1 / 1 | - / 0 / 0
include/trace/events | 97.8% / 45 / 46 | 32.9% / 24 / 73
kernel/trace | 68.2% / 11620 / 17026 | 70.6% / 1265 / 1793

Generated by: LCOV version 1.12
LCOV Examples

Source view

# of Executed

Uncovered Lines
GCOV in Userspace

To apply gcov in userspace

1. Pass “-fprofile-arcs -ftest-coverage” options to gcc when compiling a program
2. Run the program
3. You’ll see `<SOURCE>.gcda` and `<SOURCE>.gcno`
4. In the same directory, run “`gcov <SOURCE>.c`” command
5. Check generated `<SOURCE>.c.gcov`
   ○ This shows per-line execution count with source code.
LCOV in Userspace

To use lcov in userspace
1. Pass “-fprofile-arcs -ftest-coverage” options to gcc when compiling program
2. Run the program
3. You’ll see <SOURCE>.gcda and <SOURCE>.gcno
4. In the same directory, run “lcov -c -d ./ -o lcov.info”
5. Run “genhtml -o html lcov.info”
6. Open html/index.html
GCOV in Kernel

Linux kernel can export GCOV logfile via debugfs
- Pseudo GCDA files and GCNO symlinks are exported under /sys/kernel/debug/gcov/<build-path>
- Test -> Copy the pseudo logfiles (make a snapshot) -> analyze it

Enablement
- CONFIG_GCOV_KERNEL=y compiles the framework
- CONFIG_GCOV_PROFILE_ALL=y profiles the whole kernel (**not recommended**)
Subsystem Profiling by GCOV

We can enable GCOV profiling on specific subsystem or file (recommended)

Add below lines in Makefile of the subsystem

- For profiling a file (e.g. sample.c)
  GCOV_PROFILE_sample.o := y

- For profiling all files under the directory
  GCOV_PROFILE := y
Writing Function Tests with GCOV

Instructions
1. Enable GCOV_PROFILE in target subsystem and build the kernel
2. Write a simple function test
3. Run the test
4. Check GCOV result by LCOV
5. Find what is **not** covered
6. Add a new test or improve existing one
7. Goto 3 until all functions(features) are covered
Goal of Function Tests

Don’t aim to 100% coverage of lines
- Test “functions(features)” not “implementation”
- The Linux implementation is always evolving
- Do not cover critical cases (Panic, BUG, etc)

Focus on
- What functions (of code) are not executed
- Are there any possible use-case?
- Is that a “feature”?
Ftracetest Improvement

Ftrace - a collection of Linux kernel tracers
- 10 tracers + more than 1700 events + 42 options and more...
  - See /sys/kernel/debug/tracing/*
- All operations can be done via the tracefs interface (like debugfs)

Ftracetest - a collection of test cases for ftrace
- Shell-script based test framework and test cases under kselftests
  - See linux/tools/testing/selftests/ftrace/*
- Includes more than 50 test cases
- Show precise logs and summary
Ftrace test Example

Run by root user, and reported the result summary

```bash
deftrace # ./ftracetest
=== Ftrace unit tests ===
[1] Basic trace file check [PASS]
...
[68] (instance) trace_marker trigger - test snapshot trigger [PASS]
```

# of passed: 66
# of failed: 0
# of unresolved: 1
# of untested: 0
# of unsupported: 1
# of xfailed: 0
# of undefined(test bug): 0
1. Add GCOV_PROFILE := y in kernel/trace/Makefile  
   (This patch has been upstreamed, see 6b7dca401cb1)
2. Run ftracetest
   $ cd tools/testing/selftests/ftrace
   $ ./ftracetest
3. Copy GCOV data and analyze it
   $ cp -r /sys/kernel/debug/gcov/<source-dir>/linux/kernel /opt/gcov-before
   $ cd /opt/gcov-before
   $ lcov -c -d ./trace -o lcov.info && genhtml -o html lcov.info
   $ google-chrome html/kernel/trace/index.html
Let's Check Code Coverage

- 63.0% functions are covered
- 22 / 41 files are under 75% coverage of functions.
- There is room for improvement in ftrace/test

<table>
<thead>
<tr>
<th>Filename #</th>
<th>Line Coverage #</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>tracetrace.c</td>
<td>1.7 %</td>
<td>0 / 172</td>
</tr>
<tr>
<td>tracetrace.c</td>
<td>1.7 %</td>
<td>12 / 688</td>
</tr>
<tr>
<td>trace_vspace.c</td>
<td>3.7 %</td>
<td>18 / 490</td>
</tr>
<tr>
<td>trace_machswitch.c</td>
<td>9.4 %</td>
<td>16 / 159</td>
</tr>
<tr>
<td>trace_stat.c</td>
<td>20.0 %</td>
<td>27 / 125</td>
</tr>
<tr>
<td>trace_events_filter.c</td>
<td>21.7 %</td>
<td>293 / 1461</td>
</tr>
<tr>
<td>trace_events_filter_test.h</td>
<td>100.0 %</td>
<td>11 / 11</td>
</tr>
<tr>
<td>trace_events.k</td>
<td>16.3 %</td>
<td>17 / 104</td>
</tr>
<tr>
<td>trace_time.c</td>
<td>50.7 %</td>
<td>72 / 142</td>
</tr>
<tr>
<td>trace_buffer.c</td>
<td>42.1 %</td>
<td>122 / 290</td>
</tr>
<tr>
<td>trace_stats.c</td>
<td>32.5 %</td>
<td>66 / 200</td>
</tr>
<tr>
<td>trace_switch.c</td>
<td>83.7 %</td>
<td>206 / 245</td>
</tr>
<tr>
<td>trace_events.c</td>
<td>62.1 %</td>
<td>121 / 193</td>
</tr>
<tr>
<td>trace_events.c</td>
<td>55.6 %</td>
<td>1565 / 2809</td>
</tr>
<tr>
<td>trace_stat.c</td>
<td>69.2 %</td>
<td>119 / 172</td>
</tr>
<tr>
<td>trace_buf.c</td>
<td>70.0 %</td>
<td>439 / 611</td>
</tr>
<tr>
<td>trace_mem.c</td>
<td>41.7 %</td>
<td>51 / 122</td>
</tr>
<tr>
<td>trace_skb.c</td>
<td>74.0 %</td>
<td>191 / 258</td>
</tr>
<tr>
<td>trace_skb.c</td>
<td>66.7 %</td>
<td>36 / 54</td>
</tr>
<tr>
<td>trace_skb.c</td>
<td>67.4 %</td>
<td>1506 / 2234</td>
</tr>
<tr>
<td>trace_skb.c</td>
<td>63.9 %</td>
<td>158 / 263</td>
</tr>
<tr>
<td>trace_events_test.c</td>
<td>68.4 %</td>
<td>332 / 500</td>
</tr>
<tr>
<td>trace_events_test.c</td>
<td>68.9 %</td>
<td>131 / 190</td>
</tr>
<tr>
<td>trace_events_test.c</td>
<td>100.0 %</td>
<td>1 / 1</td>
</tr>
<tr>
<td>trace_events.c</td>
<td>76.1 %</td>
<td>881 / 1158</td>
</tr>
<tr>
<td>trace_events.c</td>
<td>87.2 %</td>
<td>265 / 305</td>
</tr>
<tr>
<td>trace_events.c</td>
<td>80.4 %</td>
<td>1020 / 1239</td>
</tr>
<tr>
<td>trace_events.c</td>
<td>80.0 %</td>
<td>419 / 529</td>
</tr>
<tr>
<td>trace_events.c</td>
<td>79.7 %</td>
<td>1851 / 2306</td>
</tr>
<tr>
<td>trace_events.c</td>
<td>83.6 %</td>
<td>183 / 219</td>
</tr>
<tr>
<td>trace_events.c</td>
<td>100.0 %</td>
<td>24 / 24</td>
</tr>
<tr>
<td>trace_events.c</td>
<td>100.0 %</td>
<td>2 / 2</td>
</tr>
<tr>
<td>trace_events.c</td>
<td>100.0 %</td>
<td>4 / 4</td>
</tr>
<tr>
<td>trace_events.c</td>
<td>100.0 %</td>
<td>21 / 22</td>
</tr>
<tr>
<td>trace_events.c</td>
<td>100.0 %</td>
<td>20 / 20</td>
</tr>
<tr>
<td>trace_events.c</td>
<td>91.9 %</td>
<td>66 / 72</td>
</tr>
<tr>
<td>trace_events.c</td>
<td>91.5 %</td>
<td>65 / 71</td>
</tr>
<tr>
<td>trace_events.c</td>
<td>100.0 %</td>
<td>15 / 15</td>
</tr>
<tr>
<td>trace_events.c</td>
<td>82.7 %</td>
<td>327 / 388</td>
</tr>
</tbody>
</table>
- “print_kretprobe_event()” is not tested
  - This function is for printing out the “function-return” kretprobe event
- ftracetest has a kretprobe event testcase. But it does NOT test kretprobe event “output”
Improve Test Case

Not only setting the event, but also **ensure the trace output**

```
  echo 'r:testprobe2 _do_fork $retval' > kprobe_events
  -grep testprobe2 kprobe_events
  +grep testprobe2 kprobe_events | grep -q 'arg1=$retval'
  test -d events/kprobes/testprobe2
  +
  echo 1 > events/kprobes/testprobe2/enable
  ( echo "forked")
  +
  +cat trace | grep testprobe2 | grep -q '<- _do_fork'
  +
  echo 0 > events/kprobes/testprobe2/enable
  echo ':-:testprobe2' >> kprobe_events
  clear_trace
```

(Ensure the setting is correctly done)

(Ensure the trace output)
Improvement Result

- `print_kretprobe_event()` is now tested :-)

```c
1153     goto out;
1154 42 :  trace_seq_putchar(s, '\n');
1156
1157     out:
1158 42 :  return trace_handle_return(s);
1159     }
1160
1169 static enum print_line t
1170 4 :  print_kretprobe_event(struct trace_iterator *iter, int flags,
1171                                 struct trace_event *event)
1172     {
1173     struct kretprobe_trace_entry_head *field;
1174     4 :  struct trace_seq *s = &iter->seq;
1175     struct trace_probe *tp;
1176  u8 *data;
1177     int i;
1178
1179     4 :  field = (struct kretprobe_trace_entry_head *)iter->ent;
1180     tp = container_of(event, struct trace_probe, call.event);
1181
1182     trace_seq_printf(s, "%s: (", trace_event_name(&tp->call));
1183
```
Typical Untested Patterns

Typical patterns of uncovered function-tests

- Functions that are just not touched
  - Function is documented, but not tested
  - Main function is tested, but sub options are not

- Setting without verified
  - Setting the function but just set. Not verified.
  - Not only check the result, but also verify if possible
    - set_XXX -> get_XXX
    - write_XXX -> read_XXX
    - echo 1 > XXX -> cat XXX

- Undocumented features
  - New feature is not documented, no one knows.
  - Testing a feature which will be dropped in the future
You can find the series (v3) here (https://lkml.org/lkml/2018/8/30/497)

- Add 13 new test cases
- 70.6% functions are covered
- 15 / 41 files are under 75% coverage of functions.
- Still there is room for improvement
Pitfalls

Bad signals...
- Break something (e.g. testing critical error path / panic)
  - BTW, if you find it easily, it must be a **BUG** and must be fixed soon.
- Give a stress on the system (e.g. OOM)
  - That’s a stress test.
- Start using error injection

No, you are stepping into the dark side...
Side Effects

- Improves documentation
  - docs: tracing: Add stacktrace filter command

- Orphaned functions found
  - 72809cbf ("tracing: Remove orphaned function using_ftrace_ops_list_func()")
  - 7b144b6c ("tracing: Remove orphaned function ftrace_nr_registered_ops()")

- Unused(obsoleted) features found
  - test_nop_accept/refuse are tentative function
  - hex/raw/bin output format will be replaced by trace_pipe_raw

- Real bugs :)
  - 757d9140 ("tracing/blktrace: Fix to allow setting same value")
  - Stack tracer filter doesn’t work correctly
  - GCOV kernel was broken on some arch!
Conclusion

- Using GCOV is very easy
  - For Linux kernel, you just need CONFIG_GCOV_KERNEL=y and add GCOV_PROFILE:=y
  - Show how to use gcov and lcov commands

- Function tests can be improved by GCOV
  - Easy to find untested functions
  - Explained by ftracetest case

- Ftracetest was improved by GCOV
  - ~7% coverage improved with 14 new test cases
  - Found some real bugs etc.
Future Work

- Continue to improve ftrace test
  - Check what is not tested and add new tests

- Improve other selftests
  - We can also find untested functions for other tests

- FCOV: we can use ftrace instead of GCOV for profiling “function” coverage.
  - We can dynamically change the target subsystem
  - Inline functions can be covered by kprobe dynamic event
Questions?
Thank You!!

Masami Hiramatsu <mhiramat@kernel.org>
or <masami.hiramatsu@linaro.org>