Tips for Writing Good Tests for Linux

Tim Bird
Fuego Test System Maintainer
Sr. Staff Software Engineer, Sony Electronics
Outline

• Test ecosystem problems
• Test frameworks
  • LTP
  • kselftest
  • Fuego
• Attributes of a good test
• Tips
• Resources
Test ecosystem problems

- Not enough test sharing
  - Lots of test frameworks
  - Some tests are available
    - LTP and lots of individual and benchmarks exist
    - Many tests are not shared!
- Why aren’t more aspects of QA cycle shared?
  - Many in-house tests use custom test rigs or specialized hardware
  - Interface between DUT, test system and test is not standardized
Existing Test problems

- Problems with existing Open Source tests
  - Learning curve
  - False positives
  - Useless tests
Learning curve

- For any particular test, the QA engineer must learn:
  - How to build, install and run the test
  - How to customize the test for the local environment
  - How to interpret results
- Developers need to:
  - Reproduce results
    - Have 3rd parties reproduce results
  - Report issues upstream
False positives

- Bad or missing dependencies
  - LTP tests often don’t do a good job of checking dependencies
- Some tests are too sensitive to test environment conditions
  - Extra load on the machine will cause benchmarks to behave wildly
  - Bad network, bad flash, server unavailability cause false positives
Useless tests

- Tests an attribute so basic, the test never fails
- Tests conditions that are unrelated to required behavior
- Tests conditions that are already exercised just by booting the DUT and executing the test framework
  - ex: open syscall
- Tests something rare and unlikely
  - May cost more to execute than it’s worth to find a bug
Solutions

• Need to have tests that are:
  • Well-documented
  • Easier to automate
    • Handle building and installation automatically
  • More robust
    • Handle dependencies, skip problematic tests
  • Sharable with others
    • Work in many scenarios
    • Work on many devices
    • Easily customized
Test Frameworks

- LTP
  - Linux Test Project
- kselftest
  - Kernel selftest (unit tests)
- Fuego
  - AGL/LTS test system
  - Like a test package system
LTP (Linux Test Project)

- Is a big “umbrella” project, with lots of tests
- Provides helper functions for setup, results reporting, cleanup
LTP introduction

- Mostly C and posix shell tests of kernel and core system functionality
  - No benchmarks
- Has lots of tests (>3000) in 3 broad categories
  - functional, posix conformance, realtime
  - Hard to assess coverage
    - New syscalls and behaviors show up every release
      - It’s hard to keep up
- Heavy historical focus on testing error conditions
Included test harness

- Tests can be run individually, or in groups, or stress configurations
- ltp-pan – run a named collection of tests
  - Optionally with multiple simultaneous instances
  - Optionally repeatedly
    - for a count, or
    - for a period of time
- Can customize command-line parameters
- ltprun – runs groups of tests
  - Many groups defined:
    - syscalls, input, fs, net, math, numa, etc.
    - Over 80 groups of tests
LTP output

- **Individual test results schema:**
  - TPASS – test passed (result was as expected or within tolerance)
  - TFAIL – test failed (result was unexpected or out-of-tolerance)
  - TBROK – test case broken (missing precondition, such as resource unavailable)
  - TCONF – test configuration not satisfied, such as machine type or kernel version.
  - TINFO – provides additional information about a test result
  - TWARN – provides additional information about a test condition (indicating undesirable situation), but that does not affect the test result

- **Additional meta-data from harness**
  - command line, duration, system times, exit code, etc.
LTP example test

- umount02
- Sample output:

```
tst_device.c:213: INFO: Using test device LTP_DEV='/dev/loop0'
tst_test.c:792: INFO: Timeout per run is 0h 05m 00s
tst_mkfs.c:75: INFO: Formatting /dev/loop0 with ext2 opts='' extra opts=""
mke2fs 1.42.13 (17-May-2015)
umount02.c:72: PASS: umount() fails as expected: Already mounted/busy: EBUSY
umount02.c:72: PASS: umount() fails as expected: Invalid address: EFAULT
umount02.c:72: PASS: umount() fails as expected: Directory not found: ENOENT
umount02.c:72: PASS: umount() fails as expected: Invalid device: EINVAL
umount02.c:72: PASS: umount() fails as expected: Pathname too long: ENAMETOOLONG
```

Summary:
- passed 5
- failed 0
- skipped 0
- warnings 0
Example setup & cleanup

static void setup(void)
{
    memset(long_path, 'a', PATH_MAX + 1);
    SAFE_MKFS(tst_device->dev, tst_device->fs_type, NULL, NULL);
    SAFE_MKDIR(MNTPOINT, 0775);
    SAFE_MOUNT(tst_device->dev, MNTPOINT, tst_device->fs_type, 0, NULL);
    mount_flag = 1;
    fd = SAFE_CREAT(MNTPOINT "file", 0777);
}

static void cleanup(void)
{
    if (fd > 0 && close(fd))
        tst_res(TWARN | TERRNO, "Failed to close file");
    if (mount_flag)
        tst_umount(MNTPOINT);
}
Example setup & cleanup

static void setup(void)
{
    memset(long_path, 'a', PATH_MAX + 1);
    SAFE_MKFS(tst_device->dev, tst_device->fs_type, NULL, NULL);
    SAFE_MKDIR(MNTPOINT, 0775);
    SAFE_MOUNT(tst_device->dev, MNTPOINT, tst_device->fs_type, 0, NULL);
    mount_flag = 1;
    fd = SAFE_CREAT(MNTPOINT "file", 0777);
}

static void cleanup(void)
{
    if (fd > 0 && close(fd))
        tst_res(TWARN | TERRNO, "Failed to close file");
    if (mount_flag)
        tst_umount(MNTPOINT);
}
setup and cleanup

- Use SAFE__ macros for automatic error handling
- Clean up in opposite order of resource allocation
- Use tst_* helper functions
  - There are many, to handle common operations
Example test

```c
static struct tcase {
    const char *err_desc;
    const char *mntpoint;
    int exp_errno;
} tcases[] = {
    {"Already mounted/busy", MNTPOINT, EBUSY},
    {"Invalid address", NULL, EFAULT},
    {"Directory not found", "nonexistent", ENOENT},
    {"Invalid device", "./", EINVAL},
    {"Pathname too long", long_path, ENAMETOOLONG}
};
```

```c
static void verify_umount(unsigned int n)
{
    struct tcase *tc = &tcases[n];
    TEST(umount(tc->mntpoint));
    if (TEST_RETURN != -1) {
        tst_res(TFAIL,
                "umount() succeeds unexpectedly");
        return;
    }
    if (tc->exp_errno != TEST_ERRNO) {
        tst_res(TFAIL | TTERRNO,
                "umount() should fail with %s",
                tst_strerrno(tc->exp_errno));
        return;
    }
    tst_res(TPASS | TTERRNO,
            "umount() fails as expected: %s",
            tc->err_desc);
}
```
test details

- verify_umount is the main ‘test’ routine
  - In this case, it is called with the sub-testcase number
- tst_res() is used to report results
  - Should be called once per sub-testcase (with actual result)
  - Can be called multiple times with INFO
Example struct tst_test

static struct tst_test test = {
    .tid = "umount02",
    .tcnt = ARRAY_SIZE(tcases),
    .needs_root = 1,
    .needs_tmpdir = 1,
    .needs_device = 1,
    .setup = setup,
    .cleanup = cleanup,
    .test = verify_umount,
};
struct tst_test

• Define a set of test attributes
  • Including function pointers for setup, cleanup and test
  • .tid defines the test identifier
• Can specify needed resources, which are automatically created and removed
• There is no “main” function
  • actual ‘main’ calls the routines specified in the tst_test struct.
LTP Resources

- https://github.com/Linux-test-project/ltp/wiki
  - https://github.com/linux-test-project/ltp/wiki/C-Test-Case-Tutorial

- Intro article by Cyril Hrubis (project maintainer) on LWN.net
  - https://lwn.net/Articles/625969/

- Lightning talk – Introduction and status at Fosdem 2018
  - https://fosdem.org/2018/schedule/event/linux_test_project/
LTP conclusion

- Has a lot of support for writing a good test
- LTP needs more tests, to keep it relevant
- Please add stuff to it, and fix anything you find that is broken

Some project ideas:
- Convert old tests to new API
- Document specific test cases
  - Can do this in Fuego – more on this later
- Clean up and add to developer docs
- New tests (Linux commands)
kseltest Introduction

- Is the kernel unit test framework
  - Is in the kernel source tree
    - tools/testing/selftest
- Supports local execution, or remote installation
  - Can build tarfile for installation on external DUT
  - Can cross-compile (just like kernel)
- Can select individual test sets to build or run
  - make TARGETS="size timers" kselftest
- About 350 source files in 52 directories
- Where kernel devs put their own unit tests
kselftest

- Is super-convenient if you are a kernel developer
- Does not provide a harness or helpers for setup, cleanup, common operations
- Started as ad-hoc collection of kernel subsystem unit tests
  - It’s still pretty ad-hoc...
- Is migrating to common output format
Example ksselftest test

- Sorry....
- Each test is different
- There is no “typical” example, due to lack of API
- Each one written from scratch
Output format

• TAP is preferred output format
  • Test Anything Protocol (version 13)
  • See https://testanything.org/
  • Example:

```
1..4
 ok 1 - Input file opened
not ok 2 - First line of the input valid
 ok 3 - Read the rest of the file
not ok 4 - Summarized correctly # TODO Not written yet
```

• Use ksft_* output routines, to get TAP automatically (see kselftest.h)
  • ksft_test_result_pass, ksft_test_result_fail, etc.
kselftest resources

  from Documentation/dev-tools/kselftest.rst
kselftest tips

• Don’t assume you’re building or running on the latest kernel version
  • Don’t rely on features of current kernel version
  • Allow developers of earlier kernels to run latest kselftest

• Check for dependencies at runtime and notify user if they’re not fulfilled
  • Check for root user
  • Check kernel configuration
Fuego Introduction

- Fuego =
  - host test distribution +
  - a bunch of tests + test wrappers +
  - Jenkins interface
  - ALL inside a docker container
- Is intrinsically host/target
- Fuego is like the Debian of QA software
  - A distribution of tests, each one of which can be used individually (and is maintained individually)
- About 150 test suites and benchmarks so far
Fuego test

- Is more like a packaging system than an individual test
- `fuego_test.sh` is a wrapper for:
  - build (cross-compile)
  - deploy (put on target)
  - run
  - collect results
- Can also provide a parser to:
  - Collect individual test case data
  - Create standardized output (run.json file)
  - Apply pass criteria
Fuego Architecture

- **Host machine:**
  - Volume Mount
  - Docker container:
    - Test source
    - Fuego Scripts
    - Build system
    - Jenkins
  - Toolchains
  - Config
  - Builds
  - Logs
  - Web control interface

- **Target board:**
  - Test program (deployed)
Fuego Test

- A Fuego test is usually a wrapper around an existing test:
  - Example existing tests: iozone, LTP, bonnie, iperf, Dhrystone, cyclictest
- Can also write a new individual test
  - For simple tests
  - Shell commands inside a Fuego test_run routine, or simple standalone script
- Consists of: fuego_test.sh and parser.py
- Also: spec.json, criteria.json, and other files
Fuego test example

tarball=hello-test-1.1.tgz

function test_pre_check {
    assert_define FUNCTIONAL_HELLO_WORLD_ARG
}

function test_build {
    make
}

function test_deploy {
    put hello $BOARD_TESTDIR/fuego.$TESTDIR/
}

function test_run {
    report "cd $BOARD_TESTDIR/fuego.$TESTDIR; \n    ./hello $FUNCTIONAL_HELLO_WORLD_ARG"
}

function test_processing {
    log_compare "$TESTDIR" "1" "SUCCESS" "p"
}
Fuego output

- Every test produces `run.json` file
  - test meta-data, logs, results in JSON format
- Results schema:
  - PASS
  - FAIL
  - ERROR
  - SKIP
Fuego advocacy

• Don’t write your DUT-based test in Fuego
  • I don’t care if you don’t write a Fuego test
    • I’d rather you didn’t
  • Write something for LTP or kselftest, and the whole industry benefits

• If writing a multi-node test, consider Fuego
  • Fuego supports host-client operations
    • serial, network
  • We need standard interfaces for other hardware control
  • Probably Board Control summit at Plumbers
Fuego Resources

- Fuego web server:
  - http://fuegotest.org/
  - wiki: http://fuegotest.org/wiki
- Mailing list:
  - https://lists.linuxfoundation.org/mailman/listinfo/fuego
- Repositories:
  - https://bitbucket.org/tbird20d/fuego
  - https://bitbucket.org/tbird20d/fuego-core
# Tim’s scorecard

<table>
<thead>
<tr>
<th>Attribute</th>
<th>LTP</th>
<th>kselftest</th>
<th>Fuego</th>
</tr>
</thead>
<tbody>
<tr>
<td>Well-documented</td>
<td>APIs - some tests - no</td>
<td>no</td>
<td>APIs - yes tests - in-progress</td>
</tr>
<tr>
<td>Handles builds and installs</td>
<td>yes</td>
<td>yes</td>
<td>yes+</td>
</tr>
<tr>
<td>Test scheduling</td>
<td>no</td>
<td>no</td>
<td>yes (via jenkins)</td>
</tr>
<tr>
<td>Helper routines</td>
<td>lots</td>
<td>few</td>
<td>some</td>
</tr>
<tr>
<td>(setup, cleanup, etc.)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Handles dependencies</td>
<td>some</td>
<td>no</td>
<td>lots</td>
</tr>
<tr>
<td>Customizable</td>
<td>some</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Consistent output</td>
<td>yes* (in different groups)</td>
<td>no* (TAP started)</td>
<td>yes</td>
</tr>
<tr>
<td>Test ids</td>
<td>numbers only</td>
<td>numbers only</td>
<td>some strings</td>
</tr>
<tr>
<td>Visualization</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Choosing a framework

- For white-box testing of the Linux kernel, use kselftest
- For black-box testing, use LTP
  - Especially for kernel behavior testing
- For benchmarks, extend or customize one of the current tools
  - xfstests, mmtests, iperf, etc.
- For dual-machine tests, use Fuego
  - Intrinsically supports host/target test operation
  - Needs more support for API for hardware connections (e.g. bus control, audio, video)
Tips for good tests

• Produce good output
• Make tests universal
• Avoid false positives
• Test something useful
Test output

- 6 elements of good test output:
  - Testcase identifier (tguid)
  - Description
  - Result (pass/fail)
  - Behavior
    - Expected behavior
    - Seen behavior
  - Interpretation
- Distinguish results from errors
  - Errors are problems that interfere with the test
Tips for test output

- Make results machine parsable, but human readable
  - Use unique strings for results output (e.g. TPASS)
  - Use common results schema:
    - Use the same strings to indicate:
      - pass, fail, error, skip
  - Use unique and persistent test case identifiers
  - Use line-based output
    - Output should be greppable.
  - Results exposition should follow the results or preced the results, but NOT BOTH
    - This makes the parser much easier.
Test case identifiers

• Don’t just use numbers
• TGUID = test globally unique identifier
  • LTP.syscall.umount02.03
  • LTP.syscall.umount02.try_nonexistent_dir
• Make the identifier persistent
  • That is, id should be the same run-to-run
  • BAD: list of connections is read from dynamic source, and numbers are used to indicate the network test to each one:
    • ‘net_test 1’  (= test to google.com)
    • ‘net_test 2’  (= test to amazon.com)
• Better:
  • ‘net_test connect to google’
  • ‘net_test connect to amazon’
Make tests universal

- Limit the languages used:
  - Native program or POSIX shell
- Don’t assume DUT capabilities
  - Check for dependencies
- Use minimal resources
Limited Languages

• Compiled language
  • Usually C (most common denominator)
  • Provide source, not binaries
  • Make source cross- compilable
    • Don’t assume architecture of DUT
  • Statically link, if possible
    • Avoid library dependencies

• POSIX shell
  • POSIX features only (no, not bash)
  • Use “checkbashisms” tool to find things that are unsupported by POSIX shell standard
    • Then get rid of them

• If another interpreted language, provide virtual machine with test
Use minimal resources

- Avoid dependencies, where possible
- C programs:
  - Limit usage of library calls: POSIX subset
  - Depends on the test, of course
  - OSkit defines a good minimal C library subset
    - Ignore the weird parts of memory allocation (14.5)
- Assume minimal OS features (reduced syscall set)
- Shell scripts:
  - Limit usage of external commands
    - Recommended minimum list:
      - cat, df, find, grep, free, head, mkdir, mount, ps, reboot, rm, rmdir, route, sync, tee, test, touch, true, umount, uname, uptime, xargs
  - Limit use of /proc and /sys
Detect dependencies

• When you have dependencies (and you will)...  
• Detect dependencies before test
  • Use dependency system
  • Probe system and abort early, with message
• Missing dependency = skip, not fail
  • Let user specify if a testcase should be run
  • ie Support skiplists, or auto-handle skips
Don’t assume DUT capabilities

- Don’t assume capacity or speed of DUT
  - Don’t hardcode loops or sizes
  - Automatically detect loops or sizes, if needed
    - Probe for capabilities (disk size, mem size, CPU speed)
    - Consider using a pre-test run (ie calibration run), to adjust loops or sizes
  - As a last resort, use test parameters to adjust loops or sizes
    - NOTE: test parameters are a royal pain to maintain. Please document not just their presence, but when and why they would be used
Make tests reusable

- Make tests usable in a wide variety of circumstances
  - Parameterize tests
  - Allow results criteria external to test
    - Required for benchmarks, to avoid dependency on the speed, latency, etc. of particular machine
    - Most benchmarks just produce results, but don’t evaluate them
  - Fuego allows specifying pass criteria for Benchmarks (criteria.json file)
Parameterize tests

- Parameters allow for adapting your test to circumstances
  - Should not be used as a way of avoiding writing parts of test that are difficult
  - Allows a single test to be used in different circumstances
- Parameters must be well-documented
  - This is often a big deficiency
- Use command line arguments for parameters
  - Don’t use shell environment variables
Documenting Tests

• What does it test?
• How does it test it?
• What are the expected results?
• What to do if bad results are seen?
  • What config items can be changed?
  • What /proc or /sys knobs can be adjusted?
  • What hardware can be changed? (e.g. mmc, antenna, etc.)
  • Where to report failures?
• What do parameters adjust?
Test automation

- Things that make a test automatable:
  - Uses standard build tropes (configure, make)
  - Is self-contained
  - Creates needed resources, cleans up after self
  - Has easily parsed output for results determination
    - Has consistent output patterns
  - Is deterministic
  - Does not require human setup or input
Test usability

- Things that make a test usable:
  - Indicates what it is testing
  - Gives additional information when test fails
Test robustness

- Check for dependencies
- Create needed resources at test time
  - But this can require time
- Tune for DUT capabilities
  - Capacity, speed, RT latencies
- Handle errors gracefully
- Clean up after test
Test something useful

- Test behavior that your program relies on
  - Stuff that would break your app if it changed
- Don’t just test everything in the spec
- Don’t test existing behavior if your code doesn’t rely on it
  - This just codifies that behavior
- Read *your* code, not the specs or the system code, to produce a test
- Make tests for things that broke and were fixed
  - Create regression tests
  - If it broke before, it can break again
Miscellaneous

- Use clitest for shell test automation
  - Provide a script with command and expected output
  - clitest executes command and compares results
  - See https://github.com/aurellojargas/clitest
My advice and preference

• Write new functional tests in LTP
  • Has a good test library, build system is free
  • Has consistent output schema
    • Many harnesses already parse LTP output
• For existing test, publish it and add Fuego test for your test
  • Fuego can automate it, document it, make the results sharable, and provide visualization for it
• Would like to see kselftest use the LTP test library
• Need board automation standards!
Go forth and test...

Share your tests!
Fuego