Testing Linux with the Linux Test Project

Paul Larson
Linux Technology Center
IBM
plars@us.ibm.com

Abstract

The Linux Test Project is an organization aimed at improving the Linux kernel by bringing test automation to the kernel testing effort. To meet this objective, the Linux Test Project develops test suites that run on multiple platforms for validating the reliability, robustness, and stability of the Linux kernel. The LTP test suite was designed to be easy to use, portable, and flexible enough that tests could be added without requiring the developer to use functions provided by the LTP test driver. This talk will cover what the Linux Test Project is and what we are doing to help improve Linux. I also plan to talk about the features provided by the test harness, the structure of the test cases, and how test cases can be written to contribute to the Linux Test Project.

1 Introduction

Through the years of Linux development, many people have asked the question, "What is being done to test Linux?" Historically, Linux testing efforts have been primarily informal and ad-hoc in nature. Users of Linux simply use it for their own normal purposes and report any problems they find. Little has been done to bring any organized testing effort to Linux though. This matter improved somewhat in May of 2000 when Silicon Graphics Inc. ™ introduced the first version of the Linux Test Project (LTP). Since that time, many individuals in the open source community and even companies such as IBM ®, OSDL ™, and BULL ® have contributed to the LTP.

2 Using LTP

One of the design goals of the Linux Test Project was to make it easy to use. To facilitate this, the LTP includes three scripts for executing subsets of the automated tests. They are:

- **r**unalltests.sh - runs all the automated kernel tests in sequential order
- **n**etwork.sh - runs all the automated network tests in sequential order
- **d**iskio.sh - runs the stress_floppy and stress_cdrom

The runalltests.sh can be executed with little or manual setup required by the user. Even though the script is named "runalltests" it does not really run every test in the LTP. It runs all of the completely automated tests that do not require the user to perform manual setup tasks. Destructive tests and tests that consume so many system resources that they are designed to be run independently, such as a few of the memory tests, are not included in runalltests.

The network.sh script is a group of most the network tests. These are grouped separately because additional setup is required for these tests to function correctly. Two test machines are necessary to run all of the network tests. Both machines should have the same version of LTP compiled and installed in the same location. The client machine will be the one where the network.sh script is actually executed. On the server machine, a .rhosts entry should be created for the root user to allow connections from the client machine. The following services will need to be running for successful execution of the network test suite: rlogind, ftpd, telnetd, echo (stream), fingerd, and rshd. More detailed information about

the setup for the machines running LTP may be found in the document called "How To Run the Linux Test Project(LTP) Test Suite" [RunLTP].

The diskio.sh script is a small test set that runs two io intensive tests. One of these targets the cdrom drive and the other targets the floppy drive. For the cdrom test to run, a cdrom with data on it must be inserted in the cdrom drive. For the floppy stress test to run, a blank, formatted floppy disk must be in the floppy drive.

The test driver itself is called pan. Pan can be passed a file that lists the tests to be executed, execute them, and exit with 0 if all tests passed, or with a number indicating how many tests failed. The line from runalltests.sh that executes pan looks like this:

```
${LTPROOT}/pan/pan -e -S -a $$ -n $$
                   -f ${TMP}/alltests
```

The *-e* is necessary to tell pan to exit with the number of tests that failed. By default it will ignore exit statuses, but it is generally useful to have pan run this way.

The *-D* option tells pan to run tests sequentially as they are read from the command-file. If this option is not specified, it will select tests at random to run.

The *-a $$* in the command line tells pan the name of a file to use to store the active test names, pids, and commands being executed. The $$ is used here to have it use the current pid so that a unique file is used to store this information.

The *-n $$* in the command line is a tagname by which this pan process will be known. It is required and should be unique so $$ is convenient to use again.

The *-f* option is used to tell pan the name of a command-file to execute tests from. The command-file is a text file containing one test per line. The first item on the line is the tag name of the test, by which pan will know it. Usually this should match the TCID of the test. After the tag name and a space should be the executable with any necessary arguments. These files are usually stored under the runtest directory of LTP, but in the case of runalltests, several have been concatenated together into a file called alltests.

Another useful option for pan that is not used in runalltests is *-s*. The *-s* option tell pan the number of tests to run before exiting. If 0 is used here, pan will keep executing tests until it it manually stopped.

The *-t* option can be used to specify the amount of time pan should run tests. This time can be specified in seconds, minutes, hours, or days. For instance, *-t 12h* would tell pan to stop executing tests after 12 hours.

A complete list of options for pan can be found in the man page for pan in the /doc/man1 /citeLTP-Man directory under LTP.

Tests may also be executed individually without the need for running them under pan or from a script. Once compiled, the tests are linked to under the /testcases/bin directory from the top of the LTP source tree. Testcases may be executed directly from here with any valid command line options. This is very useful when a particular test is observed to cause an error. The test can be executed alone to reproduce the error rather than waiting for the entire test suite to run.

Sometimes it is desirable to modify tests slightly for debugging purposes, or to add additional testing to them. To help make it easier to find tests, they have been organized under the testcases directory into four main categories.

- **K**ernel - Kernel related tests such as filesystems, io, ipc, memory management, scheduler, and system calls.
- **N**etwork - Network tests including tests for ipv6, multicast, nfs, rpc, sctp, and network related user commands
- **C**ommands - Tests for user level commands like those commonly used in application development such as ar, ld, ldd, nm, objdump, and size.
- **M**isc - Miscellaneous tests that do not fit into one of the other categories. Tests such as crash (an adaptation of the well-known crashme test), f00f, and a floating point math set of tests can be found in the misc directory.

Other tests that are not part of the automated test scripts previously mentioned may also be found under this directory tree. For these tests that are not

automated, the only way to run them at the time being is manually.

3 Developing Tests for LTP

The Linux Test Project was designed to be flexible enough to allow test cases to be added to it without requiring the use of any cumbersome test driver specific features. The LTP does provide a small set of functions that can be used to help with the consistency of test cases and to act as a convenience for the developer, but the driver does not require their use. Tests written to be executed under the LTP should be self-contained so that it can be executed under pan, or separately. They should be able to detect within the test itself whether or not the test passed. If the test passes, it should return 0 or anything else if the test fails. The exact nature of return codes other than 0 may be different from one test to another. Most of the tests in LTP have been written in C, but they may be written in perl, shell scripting languages, or anything else as long as appropriate return values are preserved. This flexibility allows developers to take any quick test they have written to test something, make sure it returns 0 if it passes or anything else if it doesn't, and submit it for inclusion in the LTP.

Some of the functions in LTP make use of global variables that define various aspects of the test case. Even if it is unknown whether or not these functions will be used, it is a good idea to define these variables in order to be consistent with other test cases in the LTP.

```
char *TCID="test01";
```

The TCID variable should be defined in a way similar to the example above. The convention that has been used in other test cases in the LTP is the system call name, or some other name representing the test followed by a two digit number. The TCID should be different from any other LTP test case or results may be confusing when executing all the tests in the test suite. It is also a good idea to make the TCID be the same as the name of the source code file for the test. In this example, the file name should be something like *test01.c*.

The global variable `TST_TOTAL` is of type int and should be used to specify the number of individual test cases within the test program. Each test should be associated with an output line declaring the outcome of the test case.

```
extern int Tst_count;
```

The `Tst_count` variable is used as a test case counter in the main test loop. The output functions provided by LTP use this variable to get the number of the test case currently being executed. This should be automatically incremented each pass through the test loop.

```
for (lc=0; TEST_LOOPING(lc); lc++) {
...
}
```

The main test loop is just a for loop, but it implements a macro called `TEST_LOOPING()` to control the number of iterations through the loop. Standard command line options for LTP test cases allow the user to set a certain number of iterations or an amount of time to run each test. `TEST_LOOPING()` handles making sure that the test is executed for the correct number of iterations, or for the correct amount of time.

The actual test itself should be wrapped in the `TEST()` macro. The `TEST()` macro starts by resetting errno to 0 to ensure that the correct errno is detected after the test is complete. After executing the system call passed to it, `TEST()` sets two global variables. `TEST_RETURN` is set to the return code and `TEST_ERRNO` is set to the value of errno upon return. There is also a variation of the `TEST()` macro called `TEST_VOID()` that should be used for testing system calls that return void.

Tests that require little or no manual setup are preferred. Usually setup can be performed within the test itself, or with command line options that can be passed from the execution script. If manual setup is required, the test may be left out of automated execution scripts, or grouped with other tests that have similar setup requirements such as the network tests.

Many tests require a temporary directory to store files and directories created during the test. This is especially true of filesystem tests, and tests of system calls that operate on files and directories. The `tst_tmpdir()` and `tst_rmdir()` functions provide

a convenient method of creating and cleaning up a temporary area for the test to use.

The `tst_tmpdir()` function creates a unique, temporary directory based on the first three characters of the TCID global variable. Once the directory is created, it makes it the current working directory and returns to continue execution of the test. The name of the directory created will be saved in an extern char* variable called `TESTDIR` in case it is needed by the test case, and for later removal by the `tst_rmdir()` function. If it is unable to create a unique name, unable to create the directory, or unable to change directory to the new location `tst_tmpdir()` will use `tst_brk()` to output a `BROK` message for all test cases in the test and exit via the `tst_exit()` function. Since no cleanup function will be automatically performed in this situation, `tst_tmpdir()` should only be used at the beginning of the test before any resources have been created that would require a cleanup function.

The `tst_rmdir()` function will remove the temporary directory created by a call to `tst_tmpdir()` along with any other files or directories created under the temporary directory. The `system()` function is used by `tst_rmdir()` so the test case should not perform unexpected signal handling on the `SIGCHLD` signal.

One of the biggest conveniences provided by using the LTP API is `parse_opts()`. The `parse_opts()` function provides a consistent set of useful command line options for test cases, and allows the developer to easily add more options.

```
#include "test.h"
#include "usctest.h"

char *parse_opts(int argc, char *argv[],
                 option_t option_array[],
                 void (*user_help_func)());

typedef struct {
char *option;
int  *flag;
char **arg;
} option_t;
```

Option_array must be created by the developer to contain the desired options in addition to the default ones. `User_help_func()` is a pointer to a function that will be called when the user passes `-h` to the test case. This function should display usage information for the additional options added only. If you do not wish to specify any addition command line options, `parse_opts()` should be called with NULL for *option_array* and `user_help_func()`.

The default options provided by parse_opts are:

- -c n - Fork n copies of this test and run them in parallel. If -i or -I are also specified, each forked copy will run for the given number of iterations or amount of time respectively.
- -e - Log all errnos received during the test.
- -f - Suppress messages about functional testing
- -h - Print the help message listing these default options first, then call `user_help_func()` to display help for any extra options the developer may have added.
- -i n - Run the test for n consecutive iterations. Specifying a 0 for n will cause the test to loop continuously.
- -I x - Run the test loop until x seconds have passed.
- -p - Wait to receive a `SIGUSR1` before beginning the test. `TEST_PAUSE` must be used in the test at the point you want it to wait for `SIGUSR1`.
- -P x - Delay x seconds after each iteration before starting the next one.

Another useful feature of the LTP API is that it provides functions to output results and give test status in a consistent manner, and exit the test with an exit code consistent with the results from that output. This paper will not cover all of the functions to do this but will only briefly discuss the most common ones.

All of these functions need to be passed a *ttype* that specifies the type of message that is being sent. The available values for ttype are:

- **T**PASS - Indicates that the test case had the expected result and passed
- **T**FAIL - Indicates that the test case had an unexpected result and failed
- **T**BROK - Indicates that the remaining test cases are broken and will not execute correctly because some precondition was not met such as a resource not being available.

- **T**CONF - Indicates that the test case was not written to run on the current hardware or software configuration such as machine type, or kernel version.
- **T**RETR - Indicates that the test case has been retired and should not be executed any longer.
- **T**WARN - Indicates that the test case experienced an unexpected or undesirable event that should not affect the test itself such as being unable to clean up resources after the test finished.
- **T**INFO - Specifies useful information about the status of the test that does not affect the result and does not indicate a problem.

The first result output function is `tst_resm()`.

```
void tst_resm(int ttype, char *tmesg,
[arg ...])
```

This function will output *tmesg* to `STDOUT`. The *tmesg* string and associated args can be given to `tst_resm()` and the other functions listed here in the same fashion as strings with args can be passed to `printf()`. After outputting the message, the test case will resume.

```
void tst_brkm(int ttype, void (*func)(),
char *tmesg, [arg ...])
```

The `tst_brkm()` function prints the message specified by *tmesg*, calls the function pointed to by *func*, and exits the test breaking any remaining test cases.

```
void tst_exit()
```

The `tst_exit()` function exits the test with status depending on *ttypes* passed to previous calls to functions such as `tst_brkm()` and `tst_resm()`. For `TPASS`, `TRETR`, `TINFO`, and `TCONF` the exit status is unaffected and will be 0 indicating that the test pass. `TFAIL`, `TBROK`, and `TWARN` all indicate that something went wrong during the test, or that the test failed and will cause `tst_exit()` to exit the test with a non-zero status.

When a test cases receives an unexpected signal, it is useful to provide a means of making it exit gracefully. The LTP provides a convenient way of doing this through the `tst_sig()` function.

```
#include "test.h"

void tst_sig(fork_flag, handler, cleanup)
char *fork_flag;
int (*handler)();
void (*cleanup)();
```

If the test case is creating child processes through functions such as `fork()` or `system()`, then `tst_sig` needs to know to ignore `SIGCHLD`. This can be accomplished by setting *fork_flag* to `FORK`. If the test case is not creating child processes, *fork_flag* should be set to `NOFORK`. Keep in mind that if the test uses `tst_tmpdir()` and `tst_rmdir()`, the *fork_flag* should be set to `FORK` because `tst_rmdir()` uses the `system()` library call.

The handler parameter of `tst_sig()` represents the function that will be called when an unexpected signal is intercepted. The developer may provide a custom signal handler function here that returns int, or the default signal handler may be used. To use the default signal handler for `tst_sig()`, pass `DEF_HANDLER` as the *handler* parameter to `tst_sig()`. If the default handler is used, then the *TCID* and *Tst_count* variables must be defined. The default handler will use `tst_res()` to output messages for all remaining tests that were incomplete when the signal was received.

The cleanup parameter is used to specify a cleanup function. After the handler has been executed, `tst_sig()` will execute the cleanup function. The cleanup function should take care of removing any resource used by the test such as files or directories that were created to facilitate testing. If nothing is required for cleanup, `NULL` can be passed to `tst_sig()` in place of a cleanup function.

4 The Future of LTP

Most of the future plans for the Linux Test Project focus on expanding test coverage. The majority of test cases in the LTP today test system calls. This is, of course, a very important part of testing Linux, but not the only thing that should be addressed. Some tests have already been added for things such as networking, memory management, scheduling, commands, floating point math, and databases but the breadth of test coverage should continue to expand. As the variety of tests increases, it may one

day become necessary to modularize the LTP tests into separate suites that can be executed and even downloaded separately. The LTP was reorganized to make this easier if and when it becomes desirable to do so.

The completeness of current test cases should also be analyzed and improved upon if necessary. We are currently looking at code coverage analysis tools to determine how much of the target kernel code is being executed by test cases in the LTP. As we find areas of kernel code that are not adequately covered by test cases in the LTP, test cases are written or modified to expand coverage to these areas.

Additional tests are of course critical to the test suite, but for the Linux Test Project to truly be effective, people must use it. It would be nice to see the LTP test suite run as part of the exit criteria for releasing new kernels in the stable and development trees. In addition to this, it would be useful for kernel developers to execute the test suite against patches before submitting them. The LTP test suite will not find all problems but it could reduce the number of errors in new code if used properly. If kernel developers and testers diligently submit tests for defects as they are found, the test suite could even help reduce the number of regressed defects found in Linux.

References

[LTPMan] *The Linux Test Project Man Pages* Linux Test Project. `http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/ltp/ltp/doc/` (2000)

[Howto] Nate Straz, *Linux Test Project HOWTO* Linux Test Project. `http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/ltp/ltp/doc/` (2000)

[RunLTP] Casey Abell and Robbie Williamson, *How To Run the Linux Test Project(LTP) Test Suite* Linux Test Project. `http://ltp.sourceforge.net/ltphowto.php` (2001)