Testing Linux® with the Linux Test Project

Paul Larson
Linux Technology Center
International Business Machines
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 is 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 paper covers what the Linux Test Project is and what we are doing to help improve Linux. I also plan to cover 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. This matter improved somewhat in May 2000 when Silicon Graphics Inc. (SGI™) introduced the first version of the Linux Test Project (LTP). Since that time, many individuals in the open source community as well as companies such as IBM®, OSDL™, and BULL® have contributed to the LTP.

2 LTP Test Scripts

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:

- runalltests.sh – runs all the automated kernel tests in sequential order
- network.sh – runs all the automated network tests in sequential order
- diskio.sh – runs the stress_floppy and stress_cdrom tests

The runalltests.sh script can be executed with little or no manual setup required by the user. Even though the script is named "runalltests" it does not really run every test in the LTP, but it does run almost all of them. It runs all of the non-destructive and 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 test programs, are not included in the runalltests script.

The network.sh script includes most of 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 is the one where the network.sh script is actually executed. On the server machine, a .rhosts entry must 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 "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.

3 Pan: The LTP Test Driver

The test driver for the Linux Test Project test suite is called pan. Pan can parse 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 test programs that failed. By default it will ignore exit statuses, but it is generally useful to have pan run this way.

The *-S* option tells pan to run test programs sequentially as they are read from the command file. If this option is not specified, it will select test programs 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. This file is often useful for determining which test program was running last if the test machine hangs or crashes.

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

The *-f* option is used to tell pan the name of a command file to execute test programs 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, or test case ID, of the test program. 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. These command files are a convenient way of grouping test programs together to create custom test suites.

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

The *-t* option can be used to specify the amount of time pan should run test programs. This time can be specified in seconds (s), minutes (m), hours (h), or days (d). 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 [LTPMan] directory under LTP.

4 Organization of Test Programs

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

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

- Kernel – Kernel-related tests such as filesystems, io, ipc, memory management, scheduler, and system calls
- Network – Network tests including tests for ipv6, multicast, nfs, rpc, sctp, and network related user commands
- Commands – Tests for user level commands commonly used in application development such as ar, ld, ldd, nm, objdump, and size
- Misc – Miscellaneous tests that do not fit into one of the other categories such as crash (an adaptation of the well-known crashme test), f00f, and a floating point math set of tests

Other test programs that are not part of the automated test scripts previously mentioned can also be found under this directory tree.

5 Developing Tests for LTP

The Linux Test Project was designed to be flexible enough to allow test programs to be added to it without requiring the use of any cumbersome features that are specific to a certain test driver. The LTP does provide a small set of functions that can be used to help with the consistency of test programs and to act as a convenience for the developer, but the driver does not require their use. Test programs written to be executed under the LTP should be self-contained so that they can be executed under pan or separately. They should be able to detect within the test program itself whether or not each test case passed or failed. The test program should return 0 if all test cases in it pass or anything else if any of them fail. The exact nature of the failure indicated by return codes other than 0 may be different from one test program to another. Most of the test programs in LTP are 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 program 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.

The TCID variable should be defined in a way similar to the example below:

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

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 that used by any other LTP test case or results may be confusing after 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 program. 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. A test program should output a line for each test case declaring whether the test passed or failed.

The `Tst_count` variable is used as a test case counter in the main test loop:

```
extern int Tst_count;
```

The output functions provided by LTP use `Tst_count` to get the number of the test case currently being executed. This should be automatically incremented each pass through the test loop.

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.

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

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 program. `TEST_LOOPING()` handles making sure that the test program 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 the errno variable 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()` for use with testing system calls that return void.

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

Many test programs 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 program 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 program. The name of the directory created will be saved in an extern char* variable called `TESTDIR` for possible use 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 program and exit via the `tst_exit()` function. Since no cleanup function will be performed automatically in this situation, `tst_tmpdir()` should only be used at the beginning of the test program before any resources that would require a cleanup function have been created.

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 Linux Test Project 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 additional 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 between iterations.

Another useful feature of the Linux Test Project API is that it provides functions to output results and give test status in a consistent manner, and exit the test program 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 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:

- TPASS – Indicates that the test case had the expected result and passed
- TFAIL – Indicates that the test case had an unexpected result and failed
- TBROK – 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.
- TCONF – Indicates that the test case was not written to run on the current hardware or software configuration such as machine type, or kernel version.
- TRETR – Indicates that the test case has been retired and should not be executed any longer.
- TWARN – 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.
- TINFO – 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 arguments can be given to `tst_resm()` and the other functions listed here in the same fashion as strings with arguments 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 program breaking any remaining test cases.

```
void tst_exit()
```

The `tst_exit()` function exits the test program with status depending on *ttype*s 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 passed. `TFAIL`, `TBROK`, and `TWARN` all indicate that something went wrong during the test or that the test failed. Using these values at any point in the test program before `tst_exit()` is called will cause `tst_exit()` to exit the test program 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 program 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 test cases 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 program 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.

6 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 test programs 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 test programs 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.

In addition to broader coverage, a desirable feature is a wider range of test subsets. One example of this that has often been discussed is a predefined set of test programs and different options passed to test programs to create a stress suite. This could be further subdivided into components such as a memory stress suite, a scheduler stress suite, a filesystem stress suite, and so on.

Another project for future development is a front end for defining a customized set of test programs and executing them. There is already a simple menu for launching some of the test suites under the LTP, but something more advanced is desired. Ideally, a good front end should allow the user to select from all available test programs and see a description of all of them. It should also allow the user to define how long to run the suite, how many instances to have running at once, and other options. Finally, it should allow exporting and importing of profiles so that customized test executions can be reproduced later.

Currently, functions are provided by LTP to create test cases with a consistent output format, style, and command line options. These functions are only provided for C though. Tests may be developed in other languages, or even in C, but not make use of these functions and still work under the LTP test suite. The functions are provided as a convenient way of gaining a consistent set of features found throughout most of the test programs in the LTP test suite. To encourage more test development by developers who prefer other languages and want to make use of these functions, they may be implemented in languages such as perl, python, or others upon popular request.

The LTP has recently made significant progress towards running on other architectures besides x86. The majority of test programs in the LTP test suite have been made to work on architectures such as IA64, PPC, and S390. A small amount of work may still be needed for the LTP on a few of these architectures, but the LTP test suite is executed frequently on all of these architectures. As additional equipment becomes available, this list may be expanded to include other architecture targets that are capable of running Linux. Some changes may be necessary to make the LTP test suite run cleanly on these other systems.

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, new test cases are written or existing ones are modified to expand coverage to these areas. The tools that are being developed to do this will not only allow us to see what areas of the kernel are covered by each test program, but will also allow kernel developers to find test cases that target a specific area of the kernel. So, if changes are made in the kernel and the developer wants to find test programs that will target that area of the code, they can search for the specific test programs that will do that.

7 Enhancing the LTP

Additional test programs are of course critical to the test suite, but for the Linux Test Project to be truly effective, people must use it. The Linux Test Project encourages and appreciates the contributions of anyone in the open source community who wishes to participate. 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 may reduce the number of errors in new code if used properly. If kernel developers and testers diligently submit test programs for defects as they are found, the test suite could even help reduce the number of regressed defects found in Linux.

The LTP is taking steps to encourage more community involvement. Results of testing done by the LTP are posted on the LTP website at http://ltp.sourceforge.net and on the ltp-results mailing list. Requests for testing can also be submitted on the ltp-results list. An additional mailing list exists for the purpose of discussing development of the LTP test suite. The LTP test suite is also available as a testing tool inside the STP test tool at OSDL (http://www.osdl.org/stp).

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)

Disclaimer and Trademarks

Proceedings of the
Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada