Putting LTP to test - Validating both the Linux kernel and Test-cases

Subrata Modak
Linux Technology Center, IBM INDIA,
subrata@linux.vnet.ibm.com

Balbir Singh
Linux Technology Center, IBM INDIA,
balbir@linux.vnet.ibm.com

Masatake YAMATO
Red Hat Inc.,
yamato@redhat.com

Abstract

The **Linux Test Project**(LTP)[5] is receiving renewed interest, and attention due to increased focus on testing, and integration of Linux components from several projects in the Linux Ecosystem. LTP has not only discovered bugs in the Linux kernel, but also inconsistencies between other components such as libraries, the man pages, and the kernel.

In this paper, we will cover our experience in this area and delve into the details of benefits being brought to LTP because of closer interaction with the Linux ecosystem. We will also discuss the adoption of newer technologies for static & dynamic analysis of existing test cases, and show that we can use this approach to reduce any errors in test cases (leading to better end automation of Linux testing). We will also analyze the new LTP code using various test metrics, and look at the requirements for allowing the test cases to handle errors introduced by Fault Injection. We finally propose integration of all such technologies to LTP infrastructure.

1 Introduction

The surge in growth of LTP[7] in the last couple of years has also brought along with it a host of issues that needs to be addressed immediately. While numerous patches flow in to create new test cases for the ever-expanding new features of the kernel, as well as fixing the existing ones to adapt to the enhancements of existing kernel features; one immediate and dire problem that has cropped up is to validate the quality of the test cases themselves - those of which will in turn validate the quality of the kernel code.

While the kernel code can be validated through the discovery of new bugs, but, the very question of the test case quality needs to be found and answered. Contrary to the belief that the effectiveness of a test case is limited to only finding bugs, *the importance of the test case is repeatedly established when it continuously proves the stability of the same code*. However, doubts on the quality of test cases are often raised when they fail to expose bugs for too long. There can be different reasons for this. One - the code that it is supposed to test has stabilized enough, and until somebody changes something dramatically; regression will not be discovered soon. In this scenario, although nothing can be done to the test case itself, it still retains the ability to find any regressions in future.

Another reason could be that the test cases themselves are not written as good programs. They have loopholes in the way of becoming good quality code. This is something which can be addressed effectively by analyzing the test code through various static and dynamic analysis tools available to the Open Source Community. In this paper we discuss many of them. Major issues found would be highlighted, along with the remedy to make LTP a much better project.

2 Benefits of The Linux Ecosystem

The biggest common misunderstanding about LTP may be that the relationship between LTP and other projects is one-way; LTP just tests the Linux kernel. Another misunderstanding is that the test cases in LTP can be written easily based on man pages as specification. One of the authors (YAMATO) also believed the same, before joining LTP. In reality the relationship between LTP and other projects is complimentary. This fact came to his cognizance gradually through various stages, and surprises while contributing to LTP. He has already ex-

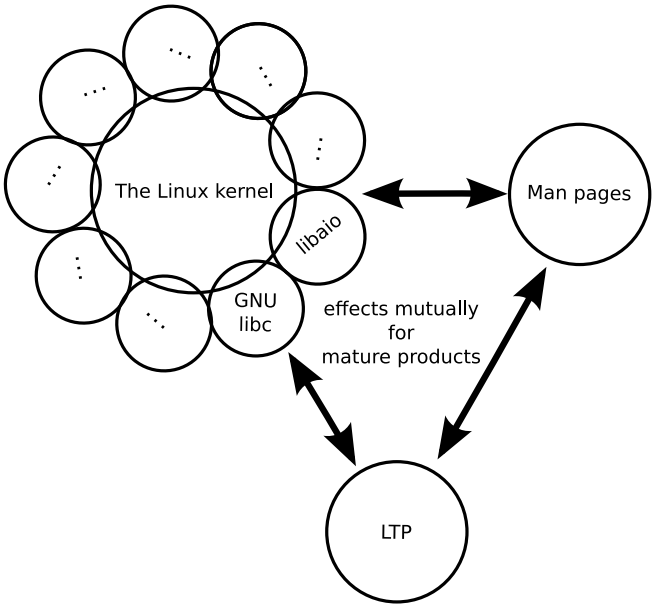


Figure 1: Linux ecosystem around test

pressed the extent of such relationship at ***Linux Ecosystem Around Test***[14].

Figure 1 shows the concept of ***The Linux Ecosystem.*** The projects in the ecosystem contribute mutually. Development efforts and/or by-products in a project are reused in other projects of the ecosystem.

2.1 Lessons learned while working on LTP

In the majority of circumstances, test cases were written based on man pages. In most instances they report **SUCCESS**. This fact leads to misunderstanding among test developers. The real worry starts only when a test case reports **FAILURE**. When FAILURE is reported for the first time, a programing error in the test case is suspected. The test case developer wonders misreading man page, and/or misusing the helper libraries of LTP. The test case **FAILURE** generates a series of investigation to find out the root cause of such a failure. And the hunt for the flaw in any component of the ecosystem starts only then.

2.1.1 Suspecting run time bug

If the error could not be found in the test case, the developer moves to the next stage. Armed with the only understanding that LTP tests just the Linux kernel, the developer suspects bug in the Linux kernel. At this point one may choose reporting the error to **Linux Kernel Mailing List**(LKML)[4] or examining the kernel source code. But this understanding alone is not enough. C language level wrapper exists between the test case and the kernel. For many system calls theses wrappers are implemented as really thin layer, and are part of GNU libc[1]. But, wrappers for some system calls do some other work, and belong to libraries other than GNU libc. These library wrapping system calls are called system libraries. Therefore before suspecting and examining Linux kernel, one must also suspect and examine the system library related to the test cases. Here, both the Linux kernel, and the system libraries as a whole are called run time entities.

The example in this section is based on author's working on a test case for `posix_fadvise()` system call[8].

When the author was working on it, the test case calls the system call with a wrong argument, and expects `EINVAL` error. The test case ran successfully on the author's PC but failed on the system used by the *bug reporter*[12].

The difference of test result came from the build configuration of kernel; The function `sys_fadvise64_64` was implemented for the system call and treated `CONFIG_EXT2_FS_XIP` configuration in wrong way[15]. `CONFIG_EXT2_FS_XIP` was *turned off* in the kernel running on the author's PC but might have been *turned on* in the kernel running on the system used by the bug reporter [1].

Only few lines were needed to fix the bug by changing the `KERNEL_CONFIG` option before building the desired kernel, however the inspection had taken rather long time. The author had inspected from upper layer: GNU libc first, and then the Linux kernel. Historically `posix_fadvise` has two variants; `posix_fadvise64` and `posix_fadvise64_64`. Their implementation look complex because many `#ifdefs` were used to share the code for the variants. Therefore the author had suspected a programming error in those `#ifdefs` causing the bug, and hence the delay.

[1]The author found "Machine Architecture: armv6l" in the bug report.

2.1.2 Suspecting man pages

With expectation that a test case for a system call can be easily based on the man pages, one may easily suspect only the run time environment, kernel and system libraries. But one would rarely suspect the man page. The activity defining the specification of new system call, and the code implementing it are not separate. In other words the system call is not implemented after its man page is written. The maintainer of man pages joins the testing[13] of the implementation of newer system call, and writes the man page only after that. Like the run time, the man pages must also be suspected.

The previous paragraph has also a real example. When the author reviewed a test case for `io_cancel` system call written by his colleague, he found strange code in it. The test case expects an error number to return when invalid arguments are passed to `io_cancel`. The strange thing was that the returned value from `io_cancel` was compared with a negative number:

```
io_cancel(ctx,NULL,NULL)==-EFAULT
```

In kernel space, negative integer is used to represent an error. But in user space, generally positive integer is used to represent an error. The test case ran successfully, but man page said the system call is expected to return positive integer when an error occurs.

There is an inconsistency between the run time and the man page for `io_cancel`. While inspecting this issue the author found (1) that the C language wrapper for `io_cancel` is not part of GNU libc, but, is part of `libaio` library. And (2) returning negative integer to represent an error is libaio's own convention. The author sent a bug report for the man pages to the man pages maintainer. As the result, the latest man pages for system calls wrapped by libaio have following NOTES:

> "GNU libc does not provide a wrapper function for this system call. The wrapper provided in libaio for `io_cancel` does not follow the usual C library conventions for indicating error: on error it returns a negated error number"

Simply to say, LTP developer has to suspect everything when test case reports **FAILURE** till it is fixed. The fix may go anywhere. And when it happens, it validates LTP's contribution to ***The Linux Ecosystem***. Good software in the **"Open Source World"** is the outcome of greater collaboration among various OSS projects. It is sometimes true for LTP that the test cases themselves are the outcome. Different from other projects, the development process itself is another outcome: finding inconsistency between run time and the man pages being a prime factor.

2.2 Role Reversal(From other projects to LTP)

The test cases for `utimensat` system call in LTP was submitted by the man pages maintainer[13]. The man pages maintainer participates in kernel development by testing and verifying newer kernel code, before the code is shipped as part of official release version of kernel. He reflects such experience when he writes man pages for them. The test code is mainly utilized three times: once for stabilizing the easy development of kernel code, another time for verifying the description of man pages, and last time for LTP. As part of LTP, the test case is run again and again on many different environments.

These days some test cases are written by kernel developers and system library developers; the original authors themselves are the test targets. The number of such test cases have increased. Some of them are imported to LTP by LTP maintainer. Some of them are directly contributed to LTP by the original authors. This is really a good trend. To write test cases, one has to understand the test target. The cost to understand the test target is the most expensive stage to write a test case. Such cost can be reduced if the original author of the test target writes test case for it.

On the other hand, writing test cases by the third person is extremely valuable; such person can have very different view to the test targets from the original authors. And this may be the strongest reason why LTP exists as a project independent from the Linux kernel, system libraries and man pages.

3 Dynamic Analysis

In the above 2 sections we explained the relationship between LTP and other projects; and introduced the proposal that we can fix bugs in the run time(the Linux kernel and support libraries) and the specification(man pages) through developing LTP test cases. However,

such kind of bug fixing can be done because LTP takes efforts to make test cases of higher quality. Any action to other projects/proposals starts from reliable test cases. Here after we will describe efforts on how we can bring more reliability to our existing test cases, by analyzing them using run time analysis tools and via static analysis tools.

3.1 Memory Leak Analysis

While there has been numerous issues reported during testing/analyzing through ***Valgrind's memcheck*** tool[11], here we look into some of the most significant ones. The following generated error clearly demonstrates that the program has been handling memory allocation/deallocation incorrectly. There has been many such instances found inside LTP, a reflection that all of them needs to be fixed to better handling of memory usage during program execution:

3.1.1 Pure Leakage Issues

A memory leak detection on the hackbench[2] tool shows

```
valgrind --leak-check=full
--show-reachable=yes
./hackbench 20 process 1000
LEAK SUMMARY:
definitely lost: 9,840 bytes in 420 blocks.
possibly lost: 0 bytes in 0 blocks.
still reachable: 3,200 bytes in
1 blocks. suppressed:
0 bytes in 0 blocks.
```

The problem *could* be addressed as shown below:

```
if(p) { free(p); (p)=NULL; }
```

at the end of every memory usage, or, before program exit would help solve such problems. And, it indeed helped. Following is the analysis post fix:

```
ERROR SUMMARY: 0 errors from 0 contexts
(suppressed: 3 from 1)
malloc/free: in use at exit:
0 bytes in 0 blocks.
malloc/free:
423 allocs, 423 frees,
14,720 bytes allocated.
All heap blocks were freed
-- no leaks are possible.
```

[2]hackbench is a part of the LTP test suite

3.1.2 Invalid Read Instances

```
Invalid read of size 8 at 0x401428:
main (clone07.c:141)
Address 0xffffffffffffff8
is not stack'd, malloc'd or (recently) free'd.
More than 10000000 total errors detected.
I'm not reporting any more.
```

The example above illustrates the example of a test case trying to reference memory, it never owned as the message shows. The run-time has been tolerant of these errors, but they are a sign of code that needs to be revisited and needs our attention.

3.1.3 Jump to Invalid Addresses

```
Jump to the invalid address stated on
the next line at 0xFFFFFFFFFF600800: ???
by 0x401650: main (getcpu01.c:125)
Address 0xfffffffffff600800 is not stack'd,
malloc'd or (recently) free'd

Process terminating with default action
of signal 11 (SIGSEGV)
Bad permissions for mapped region at
address 0xFFFFFFFFFF600800
at 0xFFFFFFFFFF600800: ???
by 0x401650:
main (getcpu01.c:125)
```

3.2 Checking Race Conditions

There are two types of tests in LTP which required to be investigated on this point. On one hand the thread tests needed analysis to see if proper synchronization exists between them, and on the other side we also wanted to see if multiple instances of the same test program(which has a single thread of execution) do not fall into the trap of deadlocks, or, race conditions over commonly accessed system resources.

3.2.1 Tests Creating Multiple Threads

This threw up some more surprises. Most of the tests falling under this category revealed that none of them has proper locking code between threads. And following output is common for all of them when analyzed through ***valgrind's helgrind***:

```
Possible data race during
write of size 8 at 0x421E508
Location 0x421E508 has never
been protected by any lock
```

Dynamic analysis shows that there is a potential race condition in the test. Submitting test results based on this test case would require further scrutiny and introspection. There is an urgent need to fix all these issues.

3.2.2 Concurrent Test Execution

Users of LTP reported that many tests inside LTP are not concurrency safe. They pointed out to various such issues and fixes for them were also proposed. We point to few of those.

- **Reserving same Port**
  The following code:

```
sin1.sin_port
 = htons(((getpid() * TST_TOTAL) % 32768)\
 + 11000 + count);
```

  is safer than the below one:

```
sin1.sin_port
 = htons((getpid() % 32768) +\
 11000 + count);
```

  if more than one process is trying to bind to the same port simultaneously, then the following error can be avoided:

```
sendfile02 2 BROK : call to bind() failed:
 Address already in use
sendfile02 3 BROK : Remaining cases broken
```

- **Generating Keys for Shared Memory Segments**

  The following comment and code snippet addresses concerns in concurrent processes who are trying to create shared memory segments concurrently.

  / Get a new IPC resource key. Since there is a small chance the getipckey() function returns the same key as the previous one, loop until we have a different key */*

```
do {
 shmkey2 = getipckey();
}while (shmkey2 == shmkey);
```

- **Using sleep() Family to Synchronize**

  Many tests use sleep() family of functions to synchronize between the parent and the child, with the hope that after a specified period of time, one will be able to have a clean access to the resource with the basic assumption that the other has already accessed it. But, this is a common programming error, and many of our old tests are victims to it. It has already been proved that such mechanism is faulty and do not provide foolproof mechanism.

  Many such instances were discovered and fixed using the mechanism of pipes to establish proper communication between parent and child and then going ahead with the desired operation.

- **The ever-famous Reader/Writer problem**

  Since most of these tests were not initially designed keeping concurrent execution in mind, they suffer from this usual design drawback. In one such instance we found test cases seem to fail when multiple instances are run concurrently. The failures occur because the file(they are trying to access) sizes don't match, or, because the number of bytes read don't match the file size. This can be attributed to one parallel instance reading a file before the other instance's write to it has completed. In such situations, either the file size has not been updated in the inode header, or, the file size has been updated, but, the file's write operation has not been updated completely. To fix this concurrency problem, we agreed to check for an existing instance and wait for it to finish before starting another instance. Any other concurrency resolution technique would complicate matters further. A message to the console indicating such a decision in scheduling policy can clarify matters cleanly.

3.3 Avoiding Segmentation faults

Certain sections of code try to access memory beyond their scope resulting in segfaults. Proper memory bounds checking before accessing/de-referencing memory will help to avoid such segmentation faults during run time. We encountered some instances of segmentation fault with LTP's provisioning engine *ltp-pan.c*. The following instance of code creates segmentation fault if `coll` is not initialized properly. De-referencing creates the problem further:

```
coll = get_collection(
 filename, optind, argc, argv);
if (coll->cnt == 0) {
```

A properly written code with checks and balances removes such faults:

```
coll = get_collection(
 filename, optind, argc, argv);
if(!coll) exit(1);
if (coll->cnt == 0)
```

3.4 Proper Exit Code

Many tests were written without proper exit code. Zeroing on all of them and fixing with appropriate return code is a big challenge given the volume of tests that exist in LTP today. Following is an excerpt of build warning generated during one such compilation:

```
hackbench.c: In function 'main':
hackbench.c:350: warning: control reaches end
 of non-void function
```

A simple `exit(RETURN_CODE)` would solve such issues and promote to better program development.

4 Static Analysis

The code that initially gave life to LTP is pretty old, and we were certain that we would hit issues that does not adhere to the latest ANSI C or good coding standards. Even if the code is to follow ANSI C Coding guidelines, still, we were faced with the dilemma of which coding pattern to follow. Being directly responsible to test the Linux kernel, we decided to go ahead with the prevalent standard in the Linux kernel community.

As a means to measure all the violations, we decided to check LTP's health with the most popular Open Source Static Analysis tools like the *SPARSE*[9] & *SPLINT*[10].

4.1 SPARSE

A single round of compilation through the code exhibited the anomalies in the program development. We would highlight few of them and probably say/decide how we can fix them.

4.1.1 Non-ANSI definitions

Numerous instances of non-ANSI definitions for various identifiers like the functions/variables were found. For example, the following definition:

```
int dataasciichk(
 listofchars, buffer, bsize,
 offset, errmsg)
char *listofchars;
char *buffer;
int bsize;
int offset;
char **errmsg) {
```

should be replaced with:

```
int dataasciichk(
 char *listofchars,
 char *buffer,
 int bsize,
 int offset,
 char **errmsg) {
```

4.1.2 Non-Static Symbol Declaration

This arose from situations where the functions and other identifiers were not defined as static although they were never used *outside* the contours of the concerned source files. The code:

```
int databinchk(...)
```

should be replaced with:

```
static int databinchk(...)
```

to avoid all such warnings. Given the volume of such messages thrown during compilation, we can definitely say that it is going to be a tough task to fix them all.

4.1.3 Symbol '*XYZ*' re-declared with different type

In older style programming as prevalent code in LTP, the general style is to declare the function prototype at the beginning of the source code, use them in different places, and then finally the definition follows at the end of the source file. Though the compilers can handle *forward references* well, still ***Sparse*** complains about it, and directs you to combine the prototype declaration and definitions together before the symbols are being referred at any point in the program.

4.1.4 Using plain integer as NULL pointer

In many places of our code, integers were directly used instead of referring them through appropriate pointers. The following code snippet:

```
sigprocmask(SIG_UNBLOCK, &newset, 0);
```

should be replaced with:

```
sigprocmask(SIG_UNBLOCK, &newset, NULL);
```

to avoid and fix such warnings.

4.1.5 Uninitialized Identifiers

This is probably the most common type of warning generated by all compilers. The safest bet would be probably to initialize them with proper values, before the undesired bug starts creeping into your program.

4.1.6 Missing type declaration for parameter '*P*'

We found some typical instances of code where a function prototype was just declared:

```
int mkname(char*, int, int);
```

But, when it came to defining that function, the type declarations for certain parameters were missing:

```
int mkname(name, me, idx)
register char   *name;
{
```

The declarations for *me* and *idx* are missing above.

4.1.7 Incompatible types for operation

These types of errors/warnings are thrown when various data types are mixed up, or, they are not properly typecaste in their respective operations. The following piece of code tries to compare whether `void *` is less then an `integer`:

```
if ((shmptr = shmat (shmid, 0, 0)) < 0)
```

4.2 SPLINT Analysis

We also found few more static cases through the *SPLINT* tool. They are really interesting enough and showed us how important programming mistakes were made during test case coding. Though many of them are safe to be ignored, still the question remains whether we should just keep ignoring them for their nature being non-fatal. This actually would reflect the concept that we were not clear about when we designed the test, leave aside a proper way to write it.

4.2.1 Return value ignored

These warnings were generated when certain sections of code were found using function calls without collecting the return value of it. The situation is inconsistent, as many other instances of code were seen collecting the function's value. The fundamental flaw is the ambiguity in designing and writing such function prototypes, when the author was not sure what to do with the function ? whether to make it return something, or, just execute a bunch of instructions.

4.2.2 Result returned by function call is not used

If there was no need for the return value of a function, why was it collected in the first place? Moreover, if the purpose is just to execute a function without the need for a return value, then the prototype could have been well defined as `void`.

4.2.3 Path with no return in function declared to return void *

Even there is something interesting. There is a path through a function declared to return a value(interestingly a `void *`) on which there is no return statement. This means the execution may fall through without returning a meaningful result to the caller.

4.2.4 Format string parameter not compile-time constant

The following piece of code should have been written like this

```
fprintf (stdout,"%s %s\n",
 global_progname, VERSION);
```

rather than this

```
char *mesg = "%s %s\n";
 fprintf (stdout, mesg,
  global_progname, VERSION);
```

If format string parameter is not a constant at compile time, then, this can lead to *security vulnerabilities* because the arguments cannot be type checked during compile time.

4.2.5 Possibility of buffer overflow

It is a commonly know fact that use of `sprintf()` has been deprecated, and/or advised to avoid. `snprintf()` is recommended instead as use of function `sprintf()` may lead to buffer overflows. However, our code base contains plenty of them and removing them would really turn out to be challenging.

4.2.6 Suspected infinite loop

Observe the following code:

```
while (child_signal_counter < num_pgrps) {
 alarm(1);
if (debug_flag >= 2)
 printf("%d: Master\
 pausing for done (%d/%d)\n", mypid,\
 child_signal_counter, num_pgrps);
pause();
}
```

No value used in loop test `(child_signal_counter,num_pgrps)` is modified by test or loop body. Hence this appears to be an infinite loop. Nothing in the body of the loop, or, the loop test modifies the value of `child_signal_counter`. Perhaps the specification of a function called in the loop body is missing a modification. Probably the only way of coming out of this loop, and hence this program is to get a signal; as probably specified by `alarm(1)`.

4.2.7 Function parameter values declared as manifest array

Though the following type of declaration is harmless

```
... compute_median (unsigned long
 values[MAX_ITERATIONS],
 unsigned long max_value);
```

as size constant is meaningless here. The size of the array is ignored in this context, since the array formal parameter is treated as a pointer. A more hassle free declaration could be just this

```
... compute_median (unsigned long
 values, unsigned long max_value);
```

5 Fault Injection Impact

The ability to alter the course of execution in the kernel through a fault induced path has long been known. The Linux kernel also have the necessary infrastructure to induce random faults in to the various parts of the kernel; thus forcing applications to expect an undesired behavior. The major advantage of using ***Fault Injection*** is to traverse those error paths of the kernel, which in normal circumstances(stable) would not have been touched.

The immediate fallout of such a scenario is an increase in the measurement of the code coverage of the kernel, as, it would guarantee to traverse the faulty path besides the actual execution path. The other advantage would directly go to the developers, who would like to test their kernel code under such varied scenarios.

Though, all these facts are well known, and has been proved by many projects in the Open Source Space, still, such an exercise has never been attempted by the LTP developers. However, even before we started to see the fall out of ***Kernel Fault Injection*** while executing LTP, we were sure that such an exercise will help us in two different ways:

- **Increase Kernel Code Coverage**[2]
- **Help Test Engineers to validate their test code under varied circumstances**

  While writing test cases for certain kernel functionalities, an engineer may test his test cases, by running it over:

  - Stable kernel, and
  - Fault Injected kernel:

    This would give him a bigger insight into his/her test behavior, and would in-fact help him to create a better test case/scenario description by uncovering bugs, if any, in his/her test code.

5.1 Experimenting with Fault Injection

We decided to use all the infrastructure provided in **linux-2.6.29** kernel[3], namely:

- `fail_io_timeout`

- `fail_make_request`
- `fail_page_alloc` &
- `failslab`

and use the following parameters of each of these infrastructure[3]:

- `probability`
- `interval`
- `times` &
- `space`

With `space` as `0`, `times` as -1 and `interval` greater than `1`, we varied the `probability` parameter for all the `fail*` subsystems. The following algorithm reflects the way the experiment was carried out:

```
start_code_coverage()
 loop (for each testcase)
  begin
   execute_testcase(inside_stable_kernel)
    begin
     insert_fault_into_kernel()
      loop X Times
       begin
        execute_testcase(inside_fault_kernel)
       end
     restore_kernel_to_normal()
    end
  end
end_code_coverage()
```

The results observed at varied *probability* values were amazing:

- **probability=100%**

  Our test provisioning engine never took off with `probability` value set at this level(100%). We knew that we cannot generate any useful data with such a system. We did not generate any code coverage data for this.

- **probability=30%**

  With `probability` value set to this level(30%), we indeed saw our tests running, but with some major flaws:

  – Failure of many tests

  Many tests failed which otherwise *pass* under normal circumstances. We traced the reasons for such failures owing to the fault in the kernel. A small snippet of `dmesg` output justified our observation. The following failure types

  ```
  <<<test_output>>>
  sh: /bin/mktemp: Cannot allocate memory
  Usage:
  mmapstress07 filename holesize e_pageskip
  sparseoff
  *holesize should be a multiple of
  pagesize
  *e_pageskip should be 1 always
  *sparseoff should be a multiple of
  pagesize
  Example: mmapstress07 myfile 4096 1 8192
  mmapstress07    1  FAIL  :  Test failed
  mmapstress07    0  WARN  :  tst_rmdir():
     TESTDIR was NULL; no removal attempted
  ```

  were accompanied by `dmesg` entries like

  ```
  FAULT_INJECTION: forcing a failure
  Pid: 30589, comm: ltp-pan Not tainted
       2.6.29-gcov #1
  Call Trace:
  [<c0698374>]should_fail+0x31f/0x3e0
  [<c0698266>]?should_fail+0x211/0x3e0
  [<c0514e5c>]?should_failslab+0x60/0x73
  [<c05123ca>]?slab_should_failslab+0x35/0x48
  ```

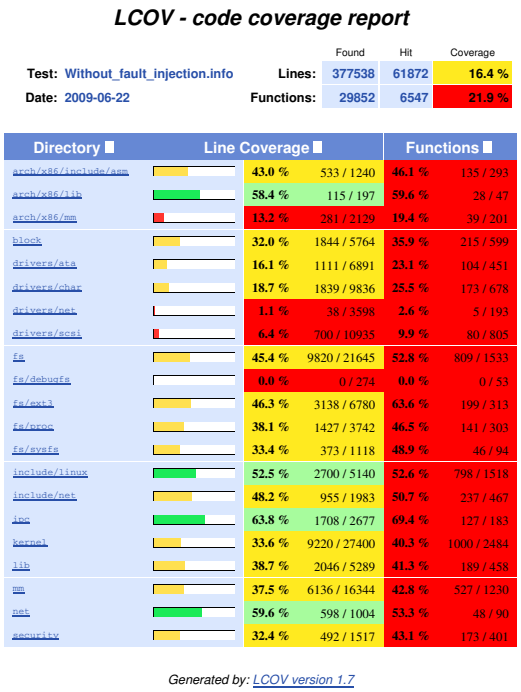
  – Long hours of execution

  Many tests took exceptionally long hours of execution time. But, otherwise, they take seconds to execute. Since, many tests in the bucket started reflecting such abnormal behavior, we had to terminate the experiment owing to the fact that the experiment cannot be continued till infinity.

- **probability=10%**

  We found this particular value more interesting; that it allowed us to run our test bucket for finite time, and simultaneously allowed us to measure the differences in the *code coverage* of test runs between the *stable* & *fault* environments. Though many tests exhibited the earlier scenarios (**probability=30%**), still they did not hinder in completing the tests in finite time frame. However, we chose a very small set of tests, namely the *LTP Syscall tests* [6].

  Figure 2 shows the code coverage obtained when the tests were run under stable kernel conditions. Out of accounted `TOTAL_CODE=377538`, **coverage** is **16.4%**, and of `TOTAL_FUNCTIONS=29852`, the tests has touched **21.9%** functions.



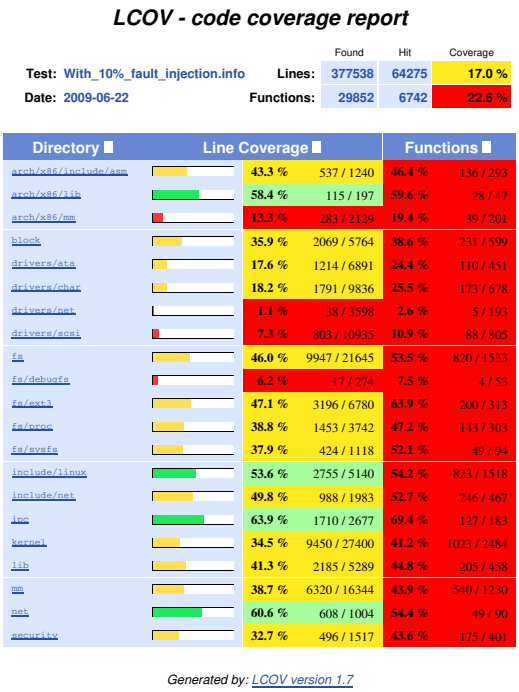
LCOV - code coverage report

		Found	Hit	Coverage
Test: Without_fault_injection.info	Lines:	377538	61872	16.4 %
Date: 2009-06-22	Functions:	29852	6547	21.9 %

Directory	Line Coverage		Functions	
arch/x86/include/asm	43.0 %	533 / 1240	46.1 %	135 / 293
arch/x86/lib	58.4 %	115 / 197	59.6 %	28 / 47
arch/x86/mm	13.2 %	281 / 2129	19.4 %	39 / 201
block	32.0 %	1844 / 5764	35.9 %	215 / 599
drivers/ata	16.1 %	1111 / 6891	23.1 %	104 / 451
drivers/char	18.7 %	1839 / 9836	25.5 %	173 / 678
drivers/net	1.1 %	38 / 3598	2.6 %	5 / 193
drivers/scsi	6.4 %	700 / 10935	9.9 %	80 / 805
fs	45.4 %	9820 / 21645	52.8 %	809 / 1533
fs/debugfs	0.0 %	0 / 274	0.0 %	0 / 53
fs/ext3	46.3 %	3138 / 6780	63.6 %	199 / 313
fs/proc	38.1 %	1427 / 3742	46.5 %	141 / 303
fs/sysfs	33.4 %	373 / 1118	48.9 %	46 / 94
include/linux	52.5 %	2700 / 5140	52.6 %	798 / 1518
include/net	48.2 %	955 / 1983	50.7 %	237 / 467
ipc	63.8 %	1708 / 2677	69.4 %	127 / 183
kernel	33.6 %	9220 / 27400	40.3 %	1000 / 2484
lib	38.7 %	2046 / 5289	41.3 %	189 / 458
mm	37.5 %	6136 / 16344	42.8 %	527 / 1230
net	59.6 %	598 / 1004	53.3 %	48 / 90
security	32.4 %	492 / 1517	43.1 %	173 / 401

Generated by: LCOV version 1.7

Figure 2: Code Coverage without Fault Injection



LCOV - code coverage report

		Found	Hit	Coverage
Test: With_10%_fault_injection.info	Lines:	377538	64275	17.0 %
Date: 2009-06-22	Functions:	29852	6742	22.6 %

Directory	Line Coverage		Functions	
arch/x86/include/asm	43.3 %	537 / 1240	46.4 %	136 / 293
arch/x86/lib	58.4 %	115 / 197	59.6 %	28 / 47
arch/x86/mm	13.3 %	283 / 2129	19.4 %	39 / 201
block	35.9 %	2069 / 5764	38.6 %	231 / 599
drivers/ata	17.6 %	1214 / 6891	24.4 %	110 / 451
drivers/char	18.2 %	1791 / 9836	25.5 %	173 / 678
drivers/net	1.1 %	38 / 3598	2.6 %	5 / 193
drivers/scsi	7.3 %	803 / 10935	10.9 %	88 / 805
fs	46.0 %	9947 / 21645	53.5 %	820 / 1533
fs/debugfs	6.2 %	17 / 274	7.5 %	4 / 53
fs/ext3	47.1 %	3196 / 6780	63.9 %	200 / 313
fs/proc	38.8 %	1453 / 3742	47.2 %	143 / 303
fs/sysfs	37.9 %	424 / 1118	52.1 %	49 / 94
include/linux	53.6 %	2755 / 5140	54.2 %	823 / 1518
include/net	49.8 %	988 / 1983	52.7 %	246 / 467
ipc	63.9 %	1710 / 2677	69.4 %	127 / 183
kernel	34.5 %	9450 / 27400	41.2 %	1023 / 2484
lib	41.3 %	2185 / 5289	44.8 %	205 / 458
mm	38.7 %	6320 / 16344	43.9 %	540 / 1230
net	60.6 %	608 / 1004	54.4 %	49 / 90
security	32.7 %	496 / 1517	43.6 %	175 / 401

Generated by: LCOV version 1.7

Figure 3: Code Coverage with Fault Injection

And Figure 3 depicts code coverage when the tests were executed under situation which is a union of stable and fault injection. Out of `TOTAL_CODE=377538`, **coverage** is **17.0%**, and of `TOTAL_FUNCTIONS=29852`, the tests has touched **22.6%** functions.

For sake of visibility & compactness, we highlight only those kernel directories & sub-directories for which significant code coverage increase has happened. Few interesting figures are:

- **3.9%** increase in **block**
- **6.2%** increase in **fs/debugfs**
- **4.5%** increase in **fs/sysfs**
- **1.2%** increase in **mm**

Though the overall increase in `CODE_COVERAGE` of **0.6%**, and `FUNCTION_COVERAGE` of **0.7%** is not significant, but it drives home a point that Code Coverage is bound to increase with `Fault Injection`. The above results are based in minimal set of LTP test cases run, and definitely the figure would be impressive, if the entire test suite is run.

6 Conclusion

The usage of Static and Dynamic Analysis tools to test LTP's health has opened up a new plethora of opportunities. These tools would be put to use more in future to validate old/new test cases. We look towards integrating them with LTP infrastructure, and they themselves becoming yardsticks for quality control. Some of the LTP test cases are beginning to show their age, they have helped identify bugs, but with newer technology and tools, it is time to revisit the test cases and shake off the bugs hiding in them, which our regular runtime execution did not expose.

Integration of Fault-Injection creation framework in LTP would be immensely beneficial to developers, who can then design robust testcases to handle these faults better. LTP also looks forward to strengthen it's position in the **Linux Ecosystem**, integrate itself with other players in the same ecosystem, so that it can continue to deliver & evolve into better test suite to *Relentlessly Pursue a Better Kernel*.

Acknowledgment

We would like to thank many of our colleagues and team mates for their inputs to, and, review of drafts of this

paper. And a special thanks to all those LTP developers whose immense contribution keeps this project growing.

Legal Statement

References

[1] Gnu c library. http://www.gnu.org/software/libc/.

[2] Lcov - the ltp gcov extension. http://ltp.sourceforge.net/coverage/lcov.php.

[3] Linux kernel documentation. http://www.mjmwired.net/kernel/Documentation/fault-injection/.

[4] Linux kernel mailing list subscription url. http://vger.kernel.org/vger-lists.html#linux-kernel.

[5] Linux test project home page. http://ltp.sourceforge.net/.

[6] Ltp source code repository. http://ltp.cvs.sourceforge.net/viewvc/ltp/ltp/runtest/syscalls.

[7] Ltp technical papers - what is ltp, how to use ltp, etc. http://ltp.sourceforge.net/documentation/technical_papers/.

[8] Man pages entry for posix_fadvise(). http://www.kernel.org/doc/man-pages/online/pages/man2/posix_fadvise.2.html.

[9] Sparse - a semantic parser for c. http://www.kernel.org/pub/software/devel/sparse/.

[10] Splint - tool for statically checking c programs for security vulnerabilities and coding mistakes. http://www.splint.org/.

[11] Valgrind. http://valgrind.org/.

[12] Pramod Gurav. [LTP] ltp tests failing. http://www.mail-archive.com/ltp-list@lists.sourceforge.net/msg00965.html.

[13] Michael Kerrisk. Linux Foundation fellowship, 6 months in. http://linux-man-pages.blogspot.com/2008/12/linux-foundation-fellowship-6-months-in.html.

[14] Masatake YAMATO. Linux ecosystem around test. http://people.redhat.com/yamato/talks/around-test.pdf.

[15] Masatake YAMATO. [PATCH] checking ADVICE of fadvice64_64 even if get_xip_page is given. http://lkml.org/lkml/2008/1/9/75.